Analysis of Congestion Models for TCP Networks

by

Robert J. Kilduff
The Hamilton Institute
National University of Ireland
Maynooth
Co. Kildare

A dissertation submitted to
the National University of Ireland
Faculty of Engineering
in partial fulfilment of the requirements
for the degree of
Masters of Engineering Science

November, 2003

Research Supervisor: Prof. Douglas Leith

Abstract

The Internet has become a global phenomena that allows people across the world to exchange information in a fast and convenient manner. In its journey to its current form, the Internet has evolved from a network research project initiated by US researchers at the Defense Advanced Research Projects Agency (DARPA) to a global network of interconnected computers. Many services which we take for granted today, such as e-mail and the World Wide Web (WWW), rely on the underlying Internet networking technologies to provide the pervasive Internet services modern society has come to expect.

The objective of this thesis is to investigate the development of mathematical models suitable for the analysis and design of TCP congestion control. While there are a multitude of protocols in use in the Internet, the most prevalent is the Transmission Control Protocol (TCP). We review current models for TCP congestion control and compare the NS packet level model (the de-facto standard for simulation studies) against real TCP implementations on a testbed network. A recently proposed simplified hybrid model with drop-tail queuing is assessed against NS and a number of modifications studied. Based on the insights gained, a simplified mathematical model of TCP congestion avoidance dynamics in synchronised networks is validated.

Dedication

To the memory of my father Sean kilduff.

Acknowledgements

This thesis is the culmination of a body of work performed while working with a research group in the Hamilton Institute. It has been an enlightening and rewarding experience. The help and advice received during this time has been far beyond my greatest expectations and I would like to take this opportunity to express my gratitude to the following people.

Within the Hamilton Institute, Prof. Douglas Leith for his advice, enthusiasm and dedication, Dr. Robert Shorten for his advice, help and encouragement, Dr. John Foy for some very useful mathematical insights and for being a good office buddy, Kai Wulff for his help with Latex, Ollie Mason for being Ollie, Miguel Vilaplana for being a crazy Spaniard, Rosemary Hunt for all the help provided and everyone else at the institute.

I would also like to thank Dr. David Malone for his help with FreeBSD and most importantly my girlfriend Margaret Burke for putting up with me during this time. Finally, I would also like to thank Science Foundation Ireland for providing the opportunity to perform this type of research in Ireland.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introductory Remarks

The internet is an evolving entity that is host to a variety of information and applications that have become pervasive to both personal and business users. The performance of the Internet is determined not only by the network and hardware technologies that underlie it, but also by the software protocols that govern its use. As the internet increases in complexity, the need for mathematical models that can be used to design such protocols, in a principled manner, is becoming more pressing. The objective of the work presented in this thesis is to contribute to this process by studying a number of packet and fluid models of networks employing TCP-like congestion control mechanisms. Motivated by the ubiquity of networks employing drop-tail queuing policies we concentrate on models of such networks and validate their predictions against measured data from real networks. A major contribution of our work is to evaluate these models from the point of view of the network dynamics; namely, to investigate those effects that most influence the dynamic behaviour of the network.

1.2 Scope of Thesis

In this thesis we consider networks that employ TCP congestion control mechanisms. Roughly speaking, TCP congestion control algorithms employ two modes of operation; Slow Start and Congestion Avoidance. The Slow Start mode of operation governs the initial evolution of data transfers whereas the Congestion Avoidance mode governs the evolution of data transfers once the Slow Start mode has been completed. Here we are chiefly concerned with the behaviour of networks whose dynamics are principally governed by the Congestion Avoidance mode of operation; these are sometimes described as long

lived flows. While such flows certainly do not dominate network traffic, they do constitute an important class of data transfers. According to studies by Brownlee et al [1] (performed in June 2002) of TCP flow duration in a typical Internet link, at least fifty five percent of flows have lifetimes of over two seconds, with two percent of flows having lifetimes of more than fifteen minutes.

The profile of Internet traffic highlights the range of conditions in which TCP is required to operate. It is clear then that models for TCP network congestion should accommodate a wide variety of data traffic profiles and a range of time scales. This is a complex requirement and to simplify the task, we will make the following assumptions about the environment that we consider. Specifically:

1. We focus on relatively long lived flows that progress through the Slow Start phase of TCP and operate in the Congestion Avoidance mode for several congestion epochs.
2. We assume that the source application always has data to send (such as a bulk FTP transfer).
3. Other flows in the network adhere to the same assumptions (no short lived flows).

For TCP, the de-facto standard model for simulation studies is NS [2]. NS is a packet level model i.e. a model that tracks individual packets as they travel across the network. Network elements are implemented as objects which are inter-connected to build up a representation of a network. One of the main features of packet level models is their complexity, which often makes it difficult to obtain the insight required for analysis and design. Here, our focus is on analysis and design and we seek models at a suitable level of abstraction that capture the key features, particularly with regard to dynamics, that are design driven. Some features that models for analysis and design might capture include the following:

- Mode changes. TCP transitions through a variety of modes including Slow Start and Congestion Avoidance, as the flow conditions change. The behaviour of the TCP source is different in these different modes.
- Entrainment. TCP sources send packets back-to-back when they have a group of packets to transmit. This can lead to packets becoming entrained with associated early overflow of queues and so called 'phase effects' [3].
- Time varying delays. Round-trip-times in networks are not fixed, depending not only on propagation delays on links but also queueing delays.

1.3 Structure of Thesis

This thesis is organised as follows. In chapter 2 we give an overview of congestion control mechanisms in the TCP protocol. In chapter 3 we compare the NS packet level model against real TCP implementations on a testbed network. In chapter 4, a simplified hybrid fluid model recently proposed by Bohacek et al [4] is compared with NS and a number of modifications proposed. Using the insight gained, a simple mathematical model of TCP congestion avoidance in synchronised networks is evaluated and analysed in chapter 5. A summary and conclusion are presented in chapter 6.

1.4 Contribution of thesis

In this theses we perform a preliminary study of modelling for the analysis and design of TCP congestion control networks. We compare the NS packet level model against a real TCP implementation and validate it under certain network conditions. During this study we highlight TCP features such as entrainment and phase effects in both NS and real TCP implementations. We report on the accuracy of NS when modelling entrainment and phase effects.

We compare the hybrid model of Bohacek et al [4] against NS and extend it to investigate discrete effects at the queue. The impact of quantisation, entrainment and back-to-back packets on the hybrid model are investigated.

We verify, through simulation, the results of a mathematical analysis of TCP congestion control in a synchronised multi-flow network. This analysis establishes the stability of TCP flows and develops criterion for fairness and convergence.

In terms of publications, the work in this thesis has led to one publication in the *Proceedings of the Twelfth Yale Workshop on Adaptive Learning and Systems* on the subject of *Towards an Analysis and Design Framework for Congestion Control in Communication Networks* [5], and a paper submitted to Automatica.

Chapter 2

An Overview of Congestion Avoidance in TCP

In this chapter we describe the features of TCP relevant to congestion control. This chapter is organised as follows. Section 2.1 provides some background information on TCP and describes the features that can exist in a typical TCP flow. In section 2.2 we look at the main TCP variants. Section 2.3 provides additional details of TCP that are relevant to our discussion. Finally section 2.4 summarises the contents of this chapter.

2.1 TCP Overview

TCP is one part of two well known protocol standards commonly referred to as TCP/IP. TCP sits on top of the IP layer and passes segments onto the IP layer for further processing. These segments are then passed onto the lower level layers and eventually onto the network. TCP was officially adopted as a standard in RFC [1] 793 [6] in 1981 and was designed to deal with message flow control and error correction, ensuring reliable delivery of a message from a source application to a destination application. IP was also officially adopted as a standard in RFC 791 [7] in 1981. IP deals with logical addressing and specifies source and destination addresses. These address are used to route a message to its destination and to provide a return address for any response.

The origins of TCP/IP stem from DARPA research into resilient networks for use in a battlefield environment [8]. The goal of this research was to design a protocol suite that could cope with network link failure and ensure delivery of data to its destination. As TCP/IP

[1] The Requests for Comments (RFC) document series is a set of technical and organizational notes pertaining to the Internet. These documents are maintained by the Internet Engineering Task Force (IETF).

evolved it moved from the research environment to be deployed in isolated networks that were eventually interconnected to become what we now know as the Internet[8].

TCP is a bi-directional, reliable, end-to-end protocol for controlling data transmission. TCP sources break messages from higher protocol layers into datagrams that are encapsulated in packets which are then transmitted over the network. These packets are reassembled by the TCP receiver into the original message and passed onto the higher level protocol layers. For every packet sent on the network by a source an acknowledgement (ACK) is expected to be transmitted back from the destination. This ACK (or lack thereof) is used by the source to determine if the acknowledged packet was successfully received at the destination. In this manner packets can be tracked and retransmitted if required.

To facilitate further discussion of TCP, the general features of TCP will be highlighted. We do not describe a specific TCP variant here but provide an overview of features a TCP variant can possess. These features will be dealt with in an historical and chronological manner. Refer to [9] for a more detailed description of the TCP protocol. In order to describe the TCP protocol some concepts are needed:

- Round Trip Time (RTT): the time taken for a packet to be sent from a source to a destination and for the corresponding acknowledgement to be received by the source, assuming no packet loss.
- Advertised window (*wnd*): the amount of data a destination has advertised that it is willing to receive. This is reported in every ACK sent by the destination to the source[2].
- Source congestion window (*cwnd*): the number of packets in flight i.e. transmitted but not yet acknowledged, assuming the source is not restricted by the advertised window.
- Send window (*swnd*): the minimum of *wnd* and *cwnd*.
- Additive Increase Multiplicative Decrease (AIMD): TCP increases its congestion window by adding to it on receipt of ACK's and decreases the send window by a multiplicative factor on receipt of an indication of packet loss.
- Slow Start: In Slow Start mode the congestion window is doubled every RTT which leads to an exponential rate of increase.
- Slow start threshold (*ssthresh*): the threshold in packets below which the source remains in slow start mode.

[2]For ease of understanding, all window sizes are described in terms of packets in this thesis. In a real TCP implementation these window sizes can be in terms of bytes.

- Fast Retransmit: a mechanism whereby the source retransmits a packet after receiving a number of duplicate ACK's (normally three) rather than waiting for a retransmit timer to timeout.
- Fast Recovery: the mode entered after a packet drop is detected via Fast Retransmit. In this mode the congestion window is restricted in value until the dropped packet is successfully retransmitted.
- Retransmission Time-Out (RTO): the interval a source waits without receiving an ACK before marking a packet as lost, retransmitting it and entering Slow Start mode. TCP calculates RTO based on current RTT and RTT variance.
- Congestion Avoidance: the mode the source enters after Fast Recovery. The source uses an AIMD strategy, linearly increasing its congestion window at a rate of one packet per RTT.
- TCP state machine: TCP is state based and maintains a local data structure to track the state of a connection. States include CLOSED, ESTABLISHED, LISTEN, and other intermediary states.
- TCP Control Block (TCB): an internal data structure that holds TCP state information and internal variables.
- Maximum Sized Segment (MSS): the maximum packet size that can be sent by a TCP source.

During the initial phase of a TCP connection the receiver (or receivers when the data flow is bi-directional) provides details of the amount of incoming data that it can process (*wnd*). This informs the source of the maximum data that the destination is currently willing to receive. In early implementations of TCP, sources transmitted data in a burst with further bursts on receipt of change in the advertised window size. This caused problems as the source tended to overwhelm the network. To address this issue Jacobson et al [10] proposed a series of measures.

Jacobson et al [10] proposed Slow Start mode. In this mode the congestion window increases on the receipt of an ACK according to the following formula

$$cwnd_{(n+1)} = cwnd_{(n)} + 1 \tag{2.1}$$

where n indexes the ACK's received. Assuming the advertised window is greater than the current *cwnd*, the source will send as many packets onto the network as allowed by its *cwnd*. The source will transmit new packets every time a new ACK is received and increments *cwnd* according to (2.1). As *cwnd* packets are sent in an RTT, the congestion window increases by *cwnd* packets per RTT. This leads to a doubling of the amount of packets sent in the next RTT causing exponential growth of *cwnd*, see for example Figure

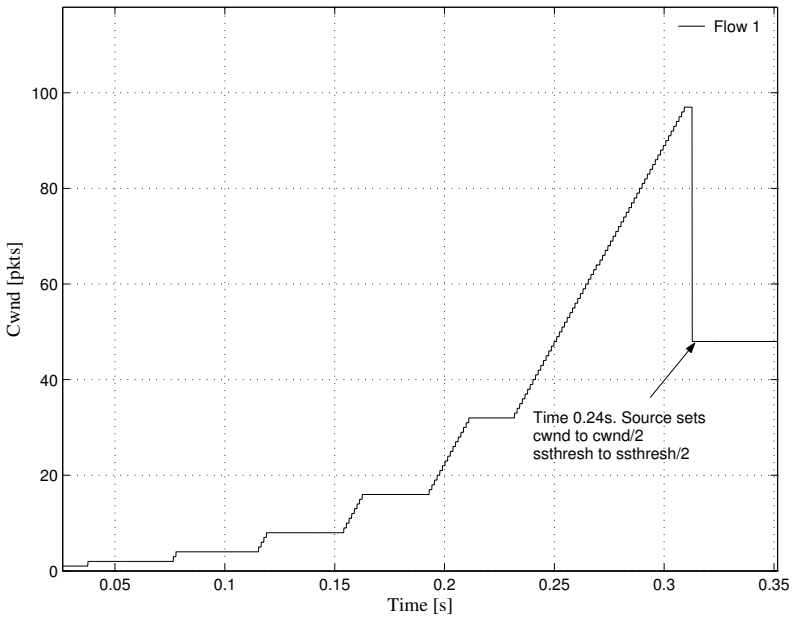

Figure 2.1: Exponential Growth of TCP Slow Start

2.1. The source increases its send window in this manner until one of two conditions is met:

1. *ssthresh* is reached. In this case the source transitions to Congestion Avoidance mode and will continue in this mode until condition 2 is met.

2. Packet loss is detected by the source. In this case the source reduces its *cwnd* by half and transitions to Fast Recovery mode.

Packet loss was originally detected at the source by the expiry of a retransmission timer. This timer is set based on current network conditions and is tied to variations in RTT. See [9] [10] for more details. This was found to be an inefficient mechanism for detection of packet loss and the Fast Retransmit mechanism was proposed to address this issue [10]. With Fast Retransmit a packet is retransmitted if three duplicate ACK's are received. This occurs as the destination only ACK's the last packet in sequence that it has received. If the destination receives packets out of order, due to a packet being dropped, it will send duplicate ACK's for the last packet received in order. The receipt of three of these duplicate ACK's is an indication of a dropped packet as detailed in RFC 2001 [11]. The source then sets *ssthresh* and *cwnd* to half its current value of *cwnd*. See, for example, time period 0.31s in Figure 2.1.

Following Fast Retransmit the source enters into Fast Recovery mode. The dropped packet(s) are retransmitted and the source remains in Fast Recovery mode until the

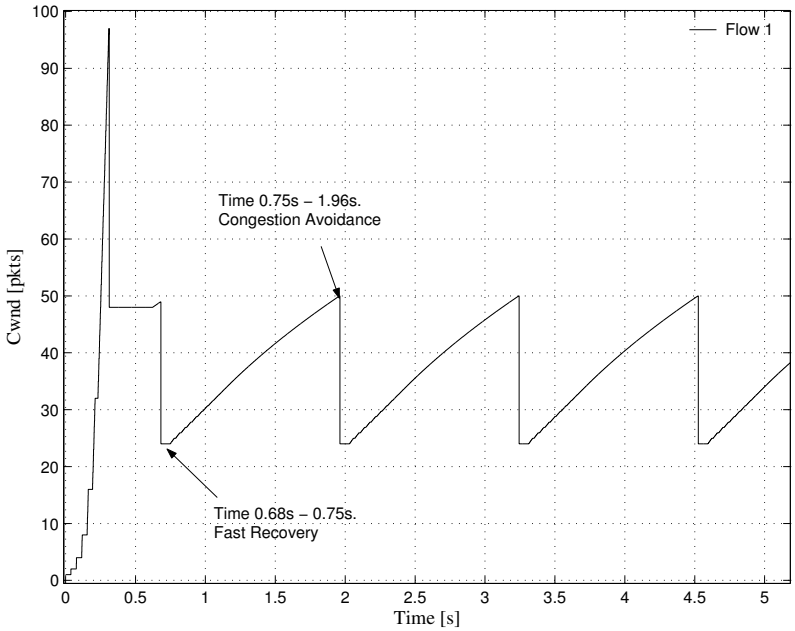

Figure 2.2: TCP Flow States

retransmitted packet is acknowledged by the destination. See time period 0.68s - 0.75s in Figure 2.2.

Following successful retransmission of lost packet(s)the source then enters into Congestion Avoidance mode. The congestion window increases according to the following formula

$$cwnd_{(n+1)} = cwnd_{(n)} + 1/cwnd_{(n)} \tag{2.2}$$

As with Slow Start the source transmits a new packet every time an ACK is received but in this mode *cwnd* is only increased by $1/cwnd$. As *cwnd* packets are sent in an RTT, *cwnd* increases by one packet per RTT. This leads to a near linear increase in *cwnd* with a slope of 1[3]. See time period 0.75s - 1.96s in Figure 2.2. The source remains in Congestion Avoidance until a packet drop is detected. In this way the TCP flow will cycle between Fast recovery and Congestion Avoidance until the end of the flow unless there is a retransmission timeout for a packet, in which case the TCP flow control will reset *cwnd* to one and restart the flow in Slow Start mode.

As the TCP source send rate is clocked by incoming ACK's the source can react to prevalent network conditions. This feedback control mechanism, called the TCP self

[3]The increase is linear in RTT but not in time as the RTT will vary over time as network queues fill and empty.

clocking mechanism, leads to a situation whereby the source increases its congestion window at a slower rate under heavy network load than under light network load.

2.2 TCP Variants

In order to understand the current status of TCP it is important to look at its development and in particular the reasoning behind specific design features. Early implementations of TCP used a go-back-n model (send sequence goes back n packets) when packets were lost. These implementations had no congestion control and led to a series of 'congestion collapses' on the Internet. During these congestion collapses the data throughput of connections was severely reduced due to excessive retransmission of packets. These issues were addressed by a version of TCP called Tahoe [10] in which the problem of congestion was approached by a 'Conservation of Packets' principle whereby new packets were not put into the network until the old ones left. Tahoe is thus a self clocking system, formed by the transmission of data and the receipt of acknowledgements.

Tahoe offered a means of combating congestion through dynamically altering the size of the protocol's send window. As can be seen in Figure 2.3 the algorithm follows these simple rules:

1. As an initial condition or if the RTO expires, set *cwnd* to one.
2. In Slow Start increase *cwnd* by one packet for each ACK until *ssthresh* is reached.
3. On reaching *ssthresh* enter Congestion Avoidance mode.
4. In Congestion Avoidance increase *cwnd* by $1/cwnd$ for each ACK received.
5. On detection of a packet loss, *cwnd* is reset to one, Slow Start is entered and *ssthresh* is set to half its current value.

Congestion Avoidance and Slow Start are independent algorithms with different objectives but in practice they are implemented together. Slow Start probes the network so that the TCP source can get an initial indication of the network bandwidth available. Congestion Avoidance more gently probes the network so that the TCP source can adapt to changing network conditions. A TCP connection will start in Slow Start mode but switch to Congestion Avoidance mode after *cwnd* reaches the value of *ssthresh*. In addition to these enhancements Tahoe also includes Fast Retransmit, better RTT variance estimation, and exponential retransmit timer back-off. These enhancements dramatically enhanced the throughput performance of TCP [12].

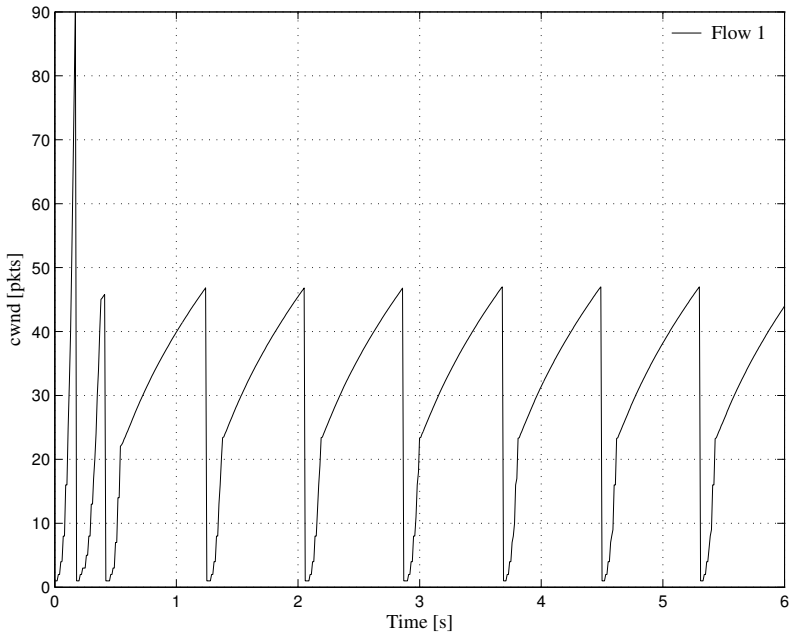

Figure 2.3: Tahoe TCP

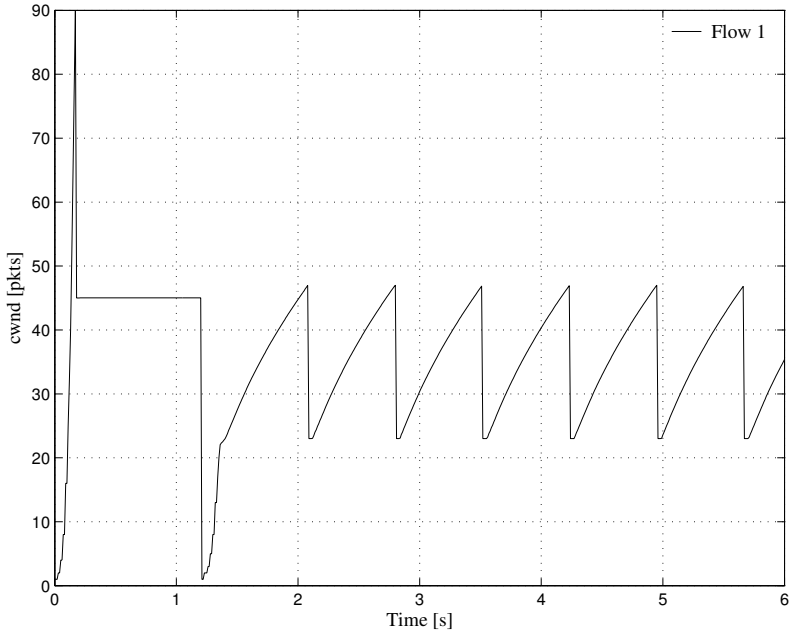

Figure 2.4: Reno TCP

Historically, the next major variant of TCP is called Reno TCP [11]. This variant of TCP is similar to the Tahoe TCP, except it also includes Fast Recovery [11]. Reno TCP does not return to Slow Start after Fast Recovery (which ends on the receipt of the retransmitted packet), instead it reduces the congestion window to half the current window size as can be seen in Figure 2.4. Note that in this example the TCP flow goes into Timeout mode following Slow Start due to excessive packet transmission during Slow Start. Reno also includes delayed ACKs which will be discussed in the next section.

TCP Tahoe and Reno experience poor performance when multiple packets are lost from one window (*cwnd*) of data[4]. With the limited information available from cumulative acknowledgments, a TCP source can only learn about a single lost packet per round trip time. An aggressive source could choose to retransmit packets early, but such retransmitted packets may have already been successfully received. TCP NewReno [13] address this issue by modifying the action taken when receiving new ACK's. In order to exit Fast Recovery, the source must receive an ACK for the highest sequence number sent before entering Fast Recovery. Thus, unlike TCP Reno, new "partial ACK's" (those which represent new ACK's but do not represent an ACK of all outstanding data) do not take TCP NewReno out of Fast Recovery. In this way, Reno retransmits one packet per RTT until all lost packets are retransmitted.

Although TCP NewReno addresses the issue of multiple drops within a window of data, it does not use all the information on dropped packets available at the receiver. This issue is addressed by the Selective Acknowledgment (SACK) mechanism [14], combined with a selective repeat retransmission policy. The receiving TCP sends back SACK packets to the source informing the source of data that has been received and any gaps that may exist due to dropped packets. The source can then retransmit packets that have been dropped. This has the benefit of allowing the source to intelligently retransmit packets and react to multiple dropped packets.

2.3 Additional TCP Details

In this section we briefly describe some additional features of TCP relevant to this thesis. To start with we look at the TCP packet format which is shown in Figure 2.5. The TCP packet comprises the following fields:

- Source Port and Destination Port. Identifies points at which upper-layer source and destination applications receive TCP services.
- Sequence Number. Specifies the number assigned to the first byte of data in the current message. In the TCP handshake phase, this field also can be used to identify

[4]Note that *cwnd* represents the number of packets in flight for the flow until a packet(s) is dropped.

an initial sequence number to be used in an upcoming transmission.

- Acknowledgment Number. Contains the sequence number of the next byte of data the source of the packet expects to receive.
- Length. Indicates the number of 32-bit words in the TCP header.
- Unused. Reserved for future use.
- Flags. Carries a variety of control information, including the SYN and ACK bits used for connection establishment, and the FIN bit used for connection termination.
- Window Size. Specifies the size of the source's receive window (that is, the buffer space available at the destination for incoming data).
- Checksum. Indicates whether the header was damaged in transit.
- Urgent Data Pointer. Points to the first urgent data byte in the packet.
- Options. Specifies various TCP options such as the MSS, the window scale value and the time the packet was sent. Also used by SACK receivers to pass lost packet information back to the source.
- Payload. Contains upper-layer information

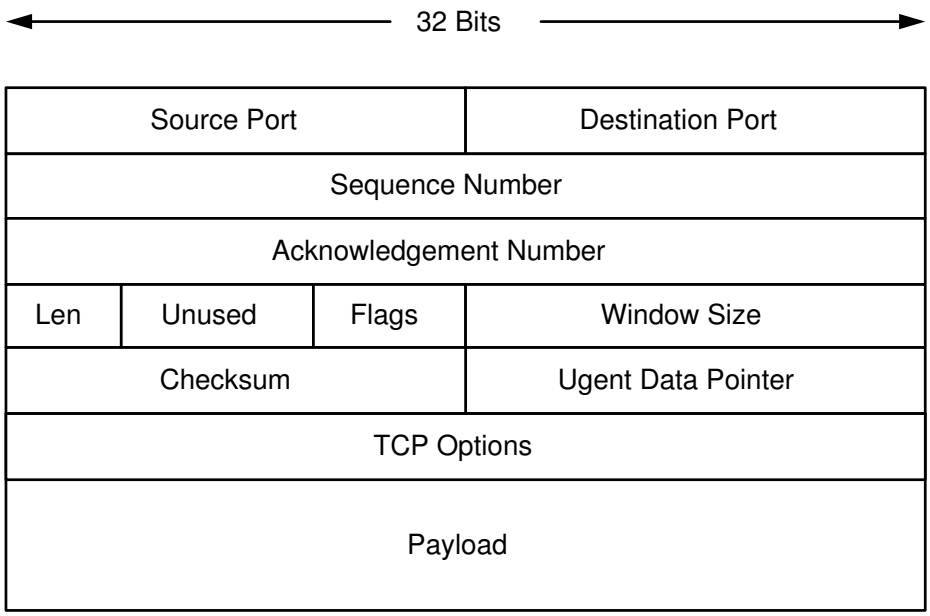

Figure 2.5: TCP Packet Format

TCP maintains an internal state machine to track the state of a TCP connection. This information is stored in the TCB and is updated as the connection changes state such as whether a connection is established or closed or in a variety of other states [9]. For example, TCP initiates a three part handshake to establish a connection with a destination. This handshake is used to set up initial conditions for the connection and works in the following way:

- The source sends a synchronisation packet (SYN) to the receiver so that its sequence numbers can be captured by the receiver and used to track incoming packets. The source's state is SYN SENT. The receiver's state is SYN RECVD.
- The receiver sends a SYN ACK packet so that its sequence numbers can be captured by the source and used to track incoming packets. From the receiver's perspective the connection is complete and its state is now ESTABLISHED.
- The source sends an ACK packet and the connection is completed. The source's state is now ESTABLISHED.

The TCB also contains per connection information such as the *cwnd*, *wnd* and the RTT. TCP uses these variables to manage flow control in order to ensure a suitable amount of data is transmitted onto the network and to reduce packet retransmission to a minimum.

In addition to congestion control the areas addressed by TCP flow control includes receiver overflow. If traffic is sent at too fast a rate for the receiver to process, its buffers may overflow and packets will need to be retransmitted. The receiver buffer size is first advertised during the initial TCP handshake and subsequently updated in every ACK transmitted to the source.

To manage a flow, TCP uses a sliding window mechanism to control the sending of packets as shown in Figure 2.6. The send window (*swnd*) slides over the data stream as ACK's are received or *wnd* changes and TCP thereby uses the send window to control the amount of packets that can be in transit in the network.

Another characteristic of TCP worthy of note deals with the unnecessary transmission of packets both from the source and the destination that affected early versions of TCP. This phenomenon known as 'Silly Window Syndrome' causes inefficient usage of the network and impacts the source and destination as they have to process a large number of small packets. On the receiver side two mechanisms are used to address this issue. The first is used after advertising a zero window. The receiver waits to transmit an ACK until a minimum of half of the receiver's buffer becomes free or it has MSS bytes ready to transmit (in this case the receiver is also a source). With the second mechanism the receiver delays sending new advertised windows until two packets have been received or for a certain time (normally 200ms). For further details see [15].

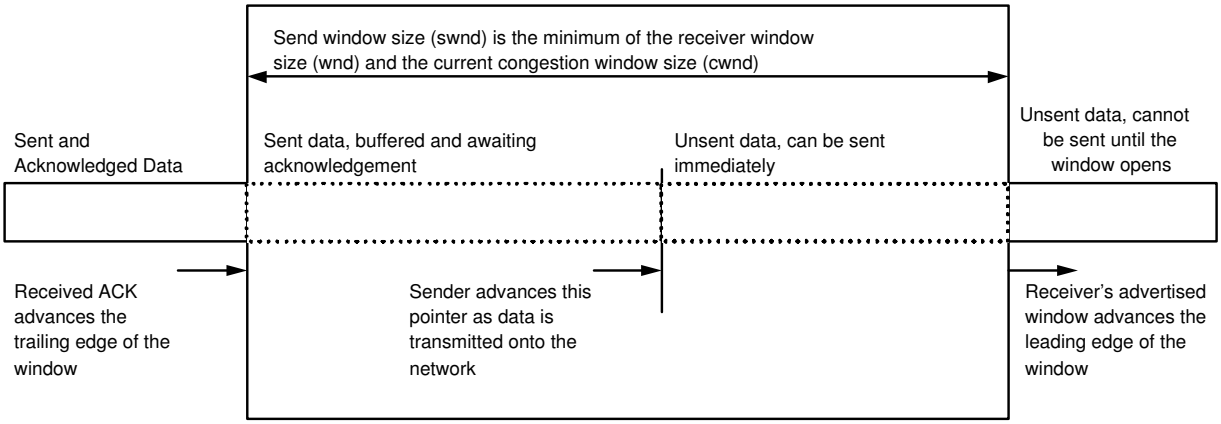

Figure 2.6: TCP sliding window[6]

On the source side, the Nagle algorithm [16] dictates when to transmit a packet. If the source has less than one MSS of data to send, a new packet will not be sent if there is any outstanding unacknowledged data. If the data to be sent is greater than one MSS, it is sent immediately. This algorithm deals with situations where applications are sending data at either slow or fast rates. When an application is sending at a slow rate data is 'clumped' and transmitted when required. This situation typically happens with interactive applications such as telnet. For applications with a requirement for faster data rates, the data is transferred in one MSS segments. This typically occurs in an application such as FTP where large data transfers are required. For more detail on both these mechanisms see [9].

The features above affect TCP network congestion analysis. In particular, for fast data rate applications, the receiver side changes cause one ACK to be sent for every two packets sent. This means that the the update rule for *cwnd*, (2.1) and (2.2), are only called at half the rate of the situation where the ACK's are not delayed. Turning this feature on or off for a TCP flow can have a large effect on its performance.

2.4 Summary

In this chapter we provided an overview of TCP congestion control. We noted that there are many variants of TCP. The main variants of TCP in use today are flavours of TCP Reno and TCP Sack. The main difference between these TCP variants lies in the manner in which they deal with lost packet recovery. For this thesis we concentrated on New Reno and Sack. We also discussed some general features of TCP such as the TCP packet

[6]This diagram is based on a paper on TCP performance by Geoff Huston [17]

format, the TCP state machine, TCP receiver overflow, the TCP sliding window and the 'Silly Window Syndrome' that are relevant to congestion control.

Chapter 3

NS Packet Level Model Experimental Validation

In this chapter we look at the NS [2] model and compare its predictions against data measured from a real network. NS is the accepted standard for modelling TCP networks but has only undergone limited validation against real networks [18]. A network consists of sources and sinks connected together via links and routers. The path between source-sink pairs consists of wireline or wireless links. We focus here on wireline links, which can be modelled as a constant propagation delay together with a queue to buffer bursty traffic. Motivated by the ubiquity of drop-tail in current networks we focus on drop-tail queues. As a complete validation of NS for TCP is beyond the scope of this thesis, we also only consider the simple dumbbell network topology (Figure 3.1) that is widely used in the research literature.

This chapter is organised as follows. In section 3.1, NS is briefly described. The hardware and software setup of our test network is detailed in section 3.2. In Section 3.3 we present a number of experiments to compare NS predictions and data from a the testbed network. Section 3.4 provides a summary of the chapter.

3.1 Overview of NS

NS is a discrete event packet level simulator. It contains a large set of components of which a subset is used for TCP. NS began as a variant of the REAL network simulator [19] in 1989 and has evolved substantially over the past few years. In 1995, NS development was supported by DARPA through the Virtual InterNetwork Testbed (VINT) project at the Lawrence Berkeley National Laboratory and has continued to be supported through various DARPA projects.

NS uses a dual software architecture which utilises both C++ and Object Tcl (OTcl). Typically OTcl is used as a frontend to the simulator where scripts can describe the simulation environment. C++ implements the lower level simulation elements such as network queues and links. Where required the class hierarchy in C++ is mirrored in OTcl so that there is a consistent view of the simulation environment from the users perspective. The logic behind the use of OTcl at the frontend is that OTcl is an interpreted language and thus suitable for use in situations where there is likely to be a large degree of change in the program. This is the case with simulation scripts. The C++ components of NS provide fixed functionality and use C++ as it provides better performance than an interpreted language. The functionality of this code can be modified by rebuilding the components but can also be modified by parameters passed through from the OTcl layer.

The NS simulator class, referenced as *ns*, contains all of the objects to be used in the simulation. These objects can be network components such as nodes, links and queues. These are defined within the simulator class and cross referenced to build up a network topology. For a detailed description of the use of these network components see the NS Manual [20].

As detailed in [20] NS models the well known variants of TCP such as Tahoe, Reno, New Reno and SACK. These objects are implemented in C++ and mirrored in the OTcl object hierarchy. Parameters specific to a TCP variant can be set at the OTcl level and bound to the relevant object or object instance at the C++ level. The core of the implementation is based on the Tahoe code. The other implementations of TCP override the Tahoe methods or add new methods as required. The code source is provided with NS and can be modified and rebuilt to add new TCP variants to NS.

A generic script for sending bulk data can be seen in Appendix A.3.1. This script was used throughout the course of this thesis and models the simple dumbbell network as detailed in Figure 3.1. The main elements of the script are:

1. the record procedure which logs the simulation data.
2. the main code that sets up the simulation environment and schedules the simulation events.
3. the finish procedure which performs post simulation tasks.

The script allows for varying the duration of the simulation, the number of sources in the simulation, the start and stop time for the sources, the duration of the link delay and the queue size for the link. The post processing script output the data in dual column format with time in the first column and simulation data in the second column.

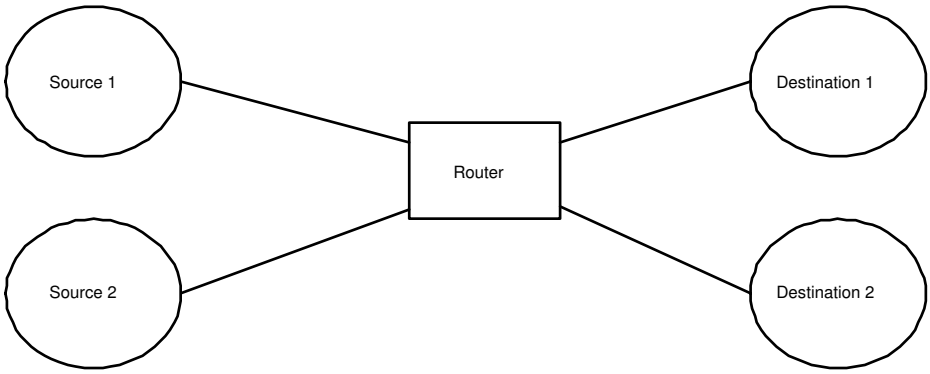

Figure 3.1: Test Network Topology

3.2 Test Environment

3.2.1 Hardware

A test network was constructed to allow experiments to be run with real TCP implementations. The test network architecture is shown in Figure 3.1. We used RFC 1918 IP addresses and the network was isolated except when experiments required access to external networks.

The two source machines and the two destination machines use the same specification:

Dell Optiplex GX250, Pentium 4, 2.0GHz
256MB RAM
Intel PRO/1000 network card
FreeBSD 4.8 RELEASE #74

The router required extra network interfaces and was the aggregation point for the network and thus a higher specification server machine was chosen.

Dell PowerEdge 1600SC, Pentium 4, 2.8GHz
2GB RAM
4xIntel PRO/1000 network card
FreeBSD 4.8 RELEASE #74

3.2.2 Source Software and Instrumentation

In order to collect measurements of internal TCP state variables such as the congestion window, it was necessary to instrument the TCP stack. The FreeBSD instrumentation

was achieved through the use of the sysctl kernel interface with logged information written to memory. A user level application then extracts the logged information.

The FreeBSD instrumentation was achieved by inserting macro code at appropriate points in the TCP input and output code. The data was logged into a structure defined in *tcp_logit.h* (see Appendix A.1.1) with elements in the following format:

```
struct tcplog {
        int type;
        struct timeval tv;
        u_short sport;
        u_short dport;
        union {
                int snd_rtt;
                int snd_srtt;
                u_long snd_cwnd;
                u_long snd_ssthresh;
                u_long snd_wnd;
                tcp_seq starttime;
        } data;
    }
```

The type field indicates the quantity stored. Valid types are RTT, Smoothed RTT, CWND, SSTHRESH, WND and STARTTIME (used to determine the time in μs when ACK's are received or new packets are sent). The next field is a timestamp in the Unix format of seconds since 1/1/1970. The next two fields hold the source port and the destination port so that a connection can be uniquely identified[1]. The next field is filled depending on the macro chosen.

Data is logged via macros defined in *tcp_logit.h*. An example for RTT is:

```
#define TCP_STASH_RTT(v,w,x) \
        do { \
                struct tcplog *l = &tcplog[tcpcurlog++ & TCPLMASK]; \
                microtime(&(l->tv)); \
                l->type = TCPL_RTT; \
                l->sport = w; \
                l->dport = x; \
                l->data.snd_rtt = v; \
        } while(0);
```

[1]As this is a simple test environment we do not need to use source and destination IP address to uniquely identify a TCP connection.

These macros are the instrumentation probes that are inserted into the FreeBSD TCP kernel code at appropriate points. Most macros are placed in the kernel code that process incoming ACK's (*tcp_input.c* and *tcp_timer.c*). The exception to this is the STARTTIME macros for sent packets which are inserted into *tcp_output.c*. The macros write to a circular memory buffer. The buffer is exposed to the user level application via the sysctl API, see *tcp_logit.c* (Appendix A.1.2).

The user level program to extract the logged kernel information is called *tcplog* (see Appendix A.1.3). This program copies the kernel buffer (using the sysctl defined above) that contains the logged data to a buffer in the user memory space. The program then cycles through the local buffer and writes the logged records out to a file. Additionally the program takes a port number as a parameter for filtering as the buffer may contain data from a number of source ports. If a port is specified on the command line *tcplog* only outputs records for that port, otherwise all records are outputted. The number of records is outputted and can be used to ensure that the circular buffer hasn't overrun.

A TCP source program called *send* was developed to provide constant TCP traffic from sources in the test network. It works by sending a stream of MSS length packets to a destination. The number of packets and the address of the destination are passed to send as command line options. The C code for *send*, *send.c* can be seen in Appendix A.2.1. The program opens a connection to the 'discard' service on the destination machine using the socket API. The 'discard' service is a service that is primarily used for test purposes. It receives packets from TCP and discards them. The program sends the number of MSS packets specified on the command line and terminates the connection, outputting the sending TCP port, the amount of time elapsed and the bits/bytes sent. The sending port is required by the *tcplog* program discussed above to uniquely identify the TCP connection.

Comment on other mechanisms to extract TCP kernel data

While we used the sysctl interface to log data, an alternative approach is to use the /proc interface. The latter is used, for example, by the web100 instrumentation patch [21] for Linux. Web100 is an implementation of an IETF Internet draft TCP MIB [22] which allows for low level instrumentation of the TCP stack.

The sysctl and /proc mechanisms to log TCP data both insert probes into the kernel code and thus are both event driven. The sysctl mechanism logs this data to memory which is copied in it entirety to user space and thus the entire contents of the logged data can be processed. The /proc mechanism in contrast requires a user level program to sample in real-time the TCP data made available via the proc kernel interface. This causes issues when a sample interval less than than 20ms is required as the the sleep call in Linux can only guarantee to wake up a process after a minimum of 20ms. This means that the finer details of changes to TCP variables may be missed. This is especially true as the transmission speed increases. Running the sampling program 'full out' with no sleep

call does not alleviate this problem as the kernel process scheduling algorithm allocates system resources to the kernel during busy periods and the logging program may thus miss important information.

3.2.3 Router Software

The router functionality was provided using *Dummynet* [23]. This emulates the effects of queues, bandwidth limitations and communication delays by modifying the existing protocol stack allowing straight forward configuration of the network environment. *Dummynet* is used in conjunction with *ipfw*, the IP firewall code in FreeBSD. *Ipfw* allows the selection of IP packets based on a combination of source and destination addresses and ports, protocol types (UDP, TCP, ICMP, ...), interface, and direction (in or out). The packet filter is programmed through a set of rules, which are applied in sequence to packets until a match is found. The rules specify actions to be taken, one of which is to forward packets to a *Dummynet* pipe. A pipe simulates the presence of a communication path, with bandwidth limitations, propagation delays, and queues.

3.3 Comparison of NS and Test Network

A series of experiments was run to compare NS with the FreeBSD TCP implementation. Delayed ACK's were turned off for the initial experiments. We also turned off caching at the sender as this causes previous values for *ssthresh* and *cwnd* to be passed through to new connections to the same destination. NS tests were run with the New Reno variant of TCP as this is the variant currently used by FreeBSD. 1500 byte packets were used throughout these experiments unless otherwise stated.

Our test network environment is isolated and so does not experience the cross traffic that exists on a real network. In addition we do not consider the effects of the host configuration (such as number of network cards, number of user/kernel TCP processes etc) on results as this is beyond the scope of this thesis.

3.3.1 Single Flow

The bottleneck link for this experiment was 10Mbit/s, queue of 17 packets and delay of 40ms. The delay is made up of 37.6ms of link delay and 1.2ms of propagation delay on the forward and backwards links. Figure 3.2 shows a comparison of the NS and FreeBSD congestion window time histories for a single flow. We note the window inflation/deflation [11] during Fast Recovery (e.g. 0.4s to 2.6s) which is not captured by NS.

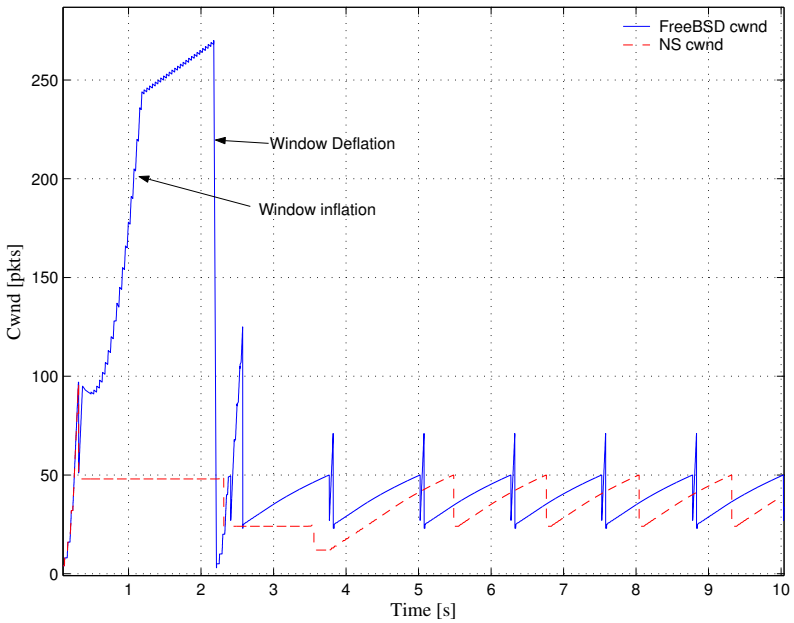

Figure 3.2: Comparison of one TCP flow using FreeBSD and NS. (10Mbit/s, 40ms delay, queue size 17 packets)

A more detailed comparison of NS and FreeBSD during Slow Start is shown in Figure 3.3. We can see an almost exact match and can clearly see rapid increases in *cwnd* followed by periods of no increase, which is a signature of packet entrainment. When a source sends a packet out onto the network, it receives an ACK one RTT later. The reception of an ACK causes the source to send more packets(which are sent back-to-back). This leads to the build up of a train of back-to-back packets on the network. Both NS and FreeBSD drop 49 packets during Slow Start. We also note that FreeBSD and NS experience different Fast Recovery times after Slow Start. This may be due to the different implementations of Fast Recovery in NS and FreeBSD.

A detailed comparison between FreeBSD and NS for Congestion Avoidance is shown in Figure 3.4. The window inflation/deflation during Fast Recovery leaves the FreeBSD flow at a higher initial value for *cwnd* and thus the flows diverge in the time axis. This can be seen more clearly in Figure 3.5 which is a close up of one period of Congestion Avoidance. One of the plots is shifted to align the time axes to allow better comparison[2]. Observe that there is a good match of slope and maximum value.

[2]We shall repeat similar time shifts in future when appropriate without reference

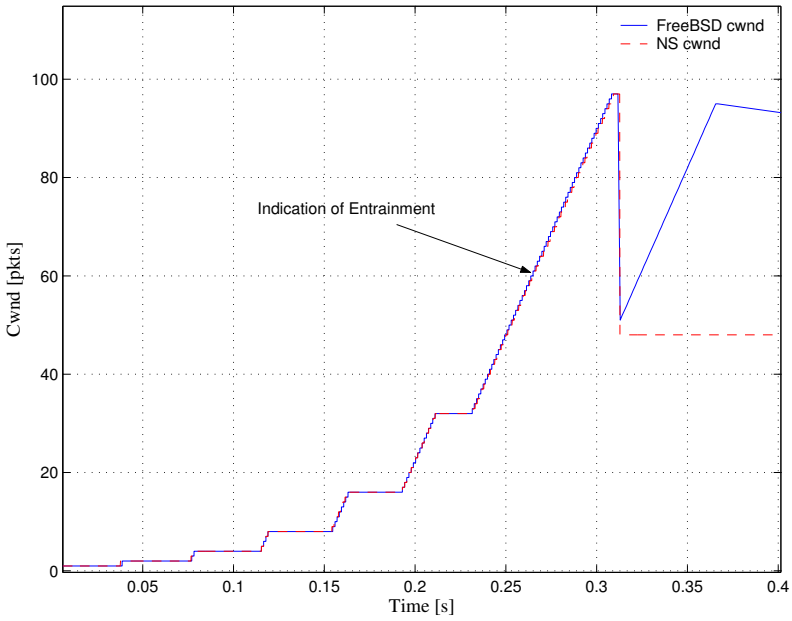

Figure 3.3: Comparison of one TCP flow during Slow Start using FreeBSD and NS (10Mbit/s, 40ms delay, queue size 17 packets)

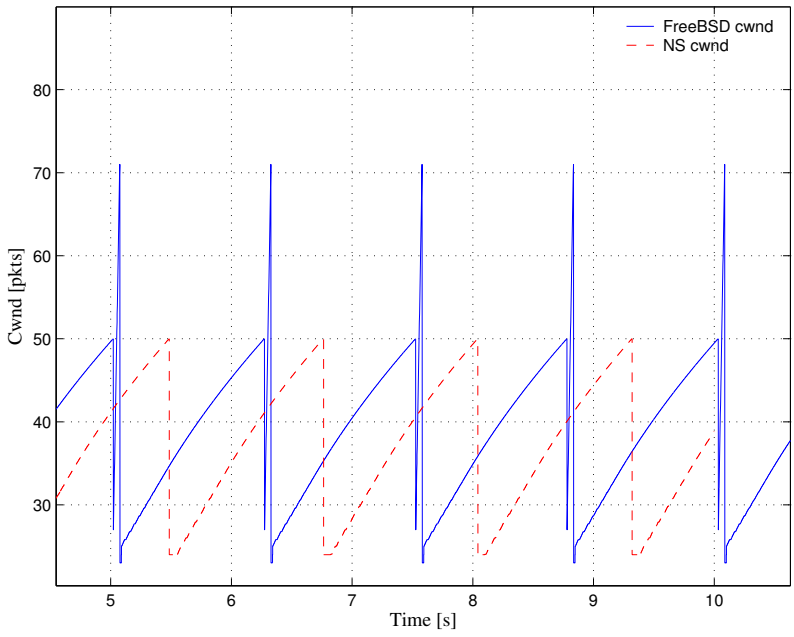

Figure 3.4: Comparison of one TCP flow during Congestion Avoidance using FreeBSD and NS (10Mbit/s, 40ms delay, queue size 17 packets)

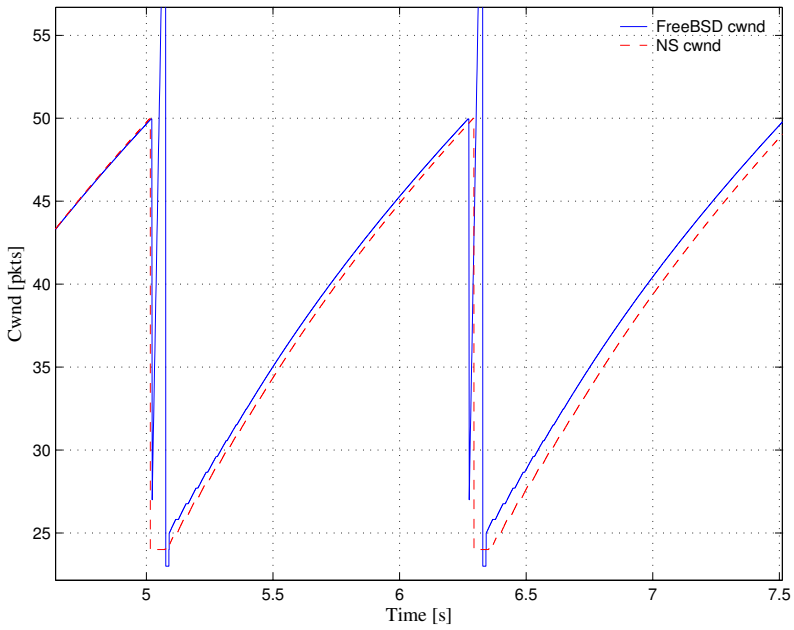

Figure 3.5: Close up of comparison of one TCP flow during Congestion Avoidance using FreeBSD and NS (10Mbit/s, 40ms delay, queue size 17 packets)

3.3.2 Two Flows

The bottleneck link for this experiment was 10Mbit/s, queue of 17 packets and delay of 40ms. The delay is made up of 37.6ms of link delay and 1.2ms of propagation delay on the forward and backwards links. All the experiments for two flows are run on the same machine and in the same process unless otherwise stated. Figure 3.6 shows a comparison of NS and FreeBSD for two flows. We note the window inflation/deflation during Fast Recovery (e.g. 0.3s to 4s) which is not captured by NS.

Figure 3.7 shows a close up of Slow Start with two flows. Again we see a good match between NS and FreeBSD. In addition we can see indications of entrainment in *cwnd* as in the single flow case but with interleaved trains of packets from each flow. The number of packet drops also matches. 48 packets were dropped in total, 32 by flow 1 and 16 by flow 2 for both NS and FreeBSD. We also note that FreeBSD and NS experience different Fast Recovery times after Slow Start. This may be due to the different implementations of Fast Recovery by NS and FreeBSD.

Figures 3.8 and 3.9 show close ups of Congestion Avoidance for FreeBSD and NS. We note that there is a good match of slope and maximum value but that the plots diverge over time due to there being different initial values for *cwnd* on exit from Fast Recovery in FreeBSD when compared to NS. We also observe the interleaved entrainment of packets

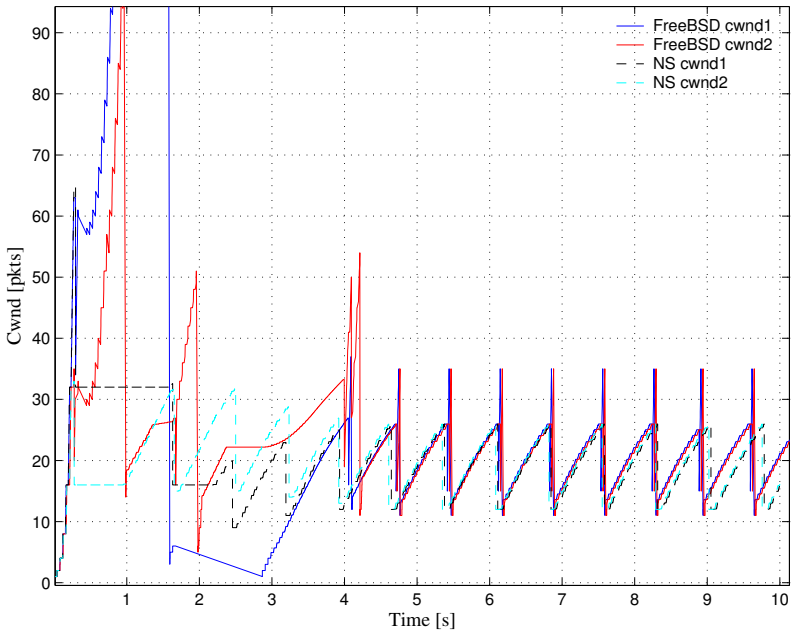

Figure 3.6: Comparison of two TCP flows using FreeBSD and NS (10Mbit/s, 40ms delay, queue size 17 packets)

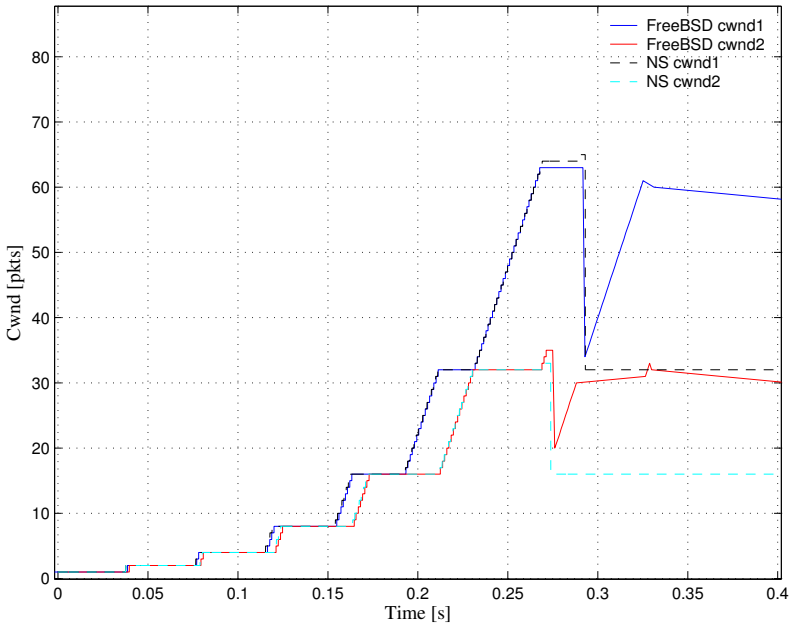

Figure 3.7: Comparison of two TCP flows during Slow Start using FreeBSD and NS (10Mbit/s, 40ms delay, queue size 17 packets)

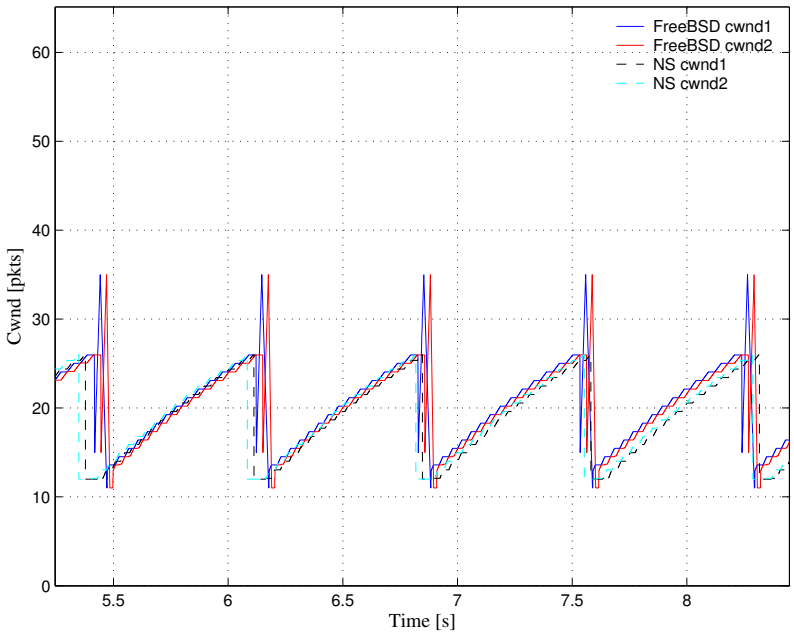

Figure 3.8: Comparison of two TCP flows during Congestion Avoidance using FreeBSD and NS (10Mbit/s, 40ms delay, queue size 17 packets)

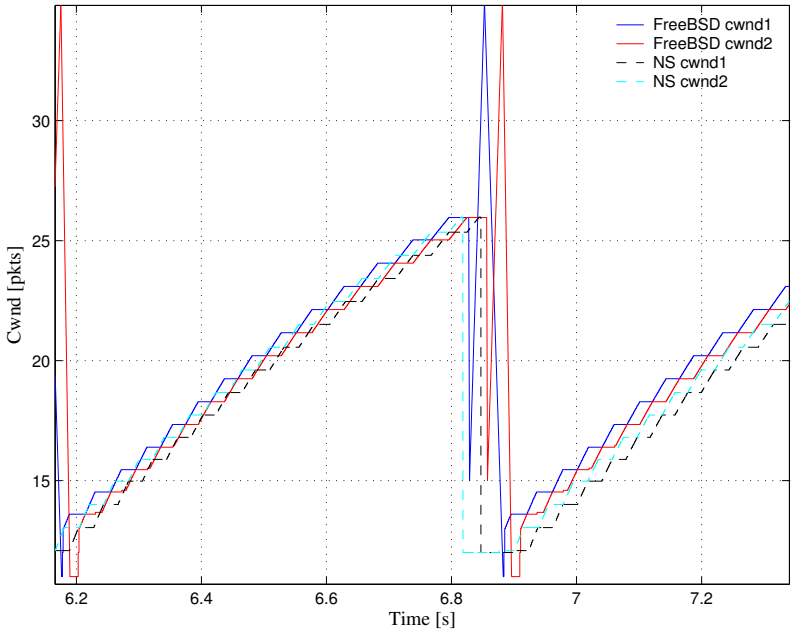

Figure 3.9: Close up of comparison of two TCP flows during Congestion Avoidance using FreeBSD and NS (10Mbit/s, 40ms delay, queue size 17 packets)

for the two flows.

3.3.3 Prevalence of Entrainment

Packet entrainment can be inferred from the results of the previous experiments for one and two flows. We see this through the time history of *cwnd* as updates to *cwnd* reflect incoming ACK's. However, while entrainment appears to exist on our testbed network, it is unclear whether it might exist on networks with a more complex topology and mix of traffic. We therefore performed experiments to determine if entrainment does indeed exist on a live production network. We used a machine located in Trinity College Dublin running FreeBSD 4.8 as the destination with delayed ACK's turned off. Our testbed was connected to the NUI Maynooth campus network via the testbed router. A traceroute from the NUI Maynooth testbed network to the Trinity College Dublin destination machine was as follows:

```
Tracing route to salmon.maths.tcd.ie [134.226.81.11] over a
maximum of 30 hops:

  1    <1 ms    <1 ms    <1 ms  10.220.3.1
  2    <1 ms    <1 ms    <1 ms  cismay.may.ie [149.157.1.6]
  3     2 ms     1 ms     1 ms  mantova-atm3-1-26.bh.access.hea.net
                                 [193.1.194.21]
  4     2 ms     2 ms     2 ms  tcd-site-gige2-0.tcd.access.hea.net
                                 [193.1.196.150]
  5     2 ms     2 ms     2 ms  tcd-hsrp187.tcd.client.hea.net
                                 [193.1.192.187]
  6     2 ms     2 ms     2 ms  mathgate.tcd.ie [134.226.10.51]
  7     2 ms     2 ms     2 ms  salmon.maths.tcd.ie [134.226.81.11]
```

Note that this route included the traversal of a number of firewalls and the Dublin internet exchange. Figure 3.10 shows sample time histories for two flows during Slow Start running to the external destination. This experiment was run at 17:54 on 3/11/2003 (Monday) and would have experienced a moderate level of cross traffic. We can see that the packets are still significantly entrained. Figure 3.11 shows a close up of the *cwnd* time histories during Congestion avoidance. Again the presence of significant entrainment can be observed.

In addition to the foregoing tests, further experiments were carried out with an artificial delay inserted into the network using *dummynet*. This experiment was run at 18:26 on 3/11/2003 and would have experienced a moderate level of cross traffic. Figure 3.12 shows the results for two flows during Slow Start running to the external destination with a delay of 40ms. Figure 3.13 show the results during Congestion Avoidance. Evidently, significant

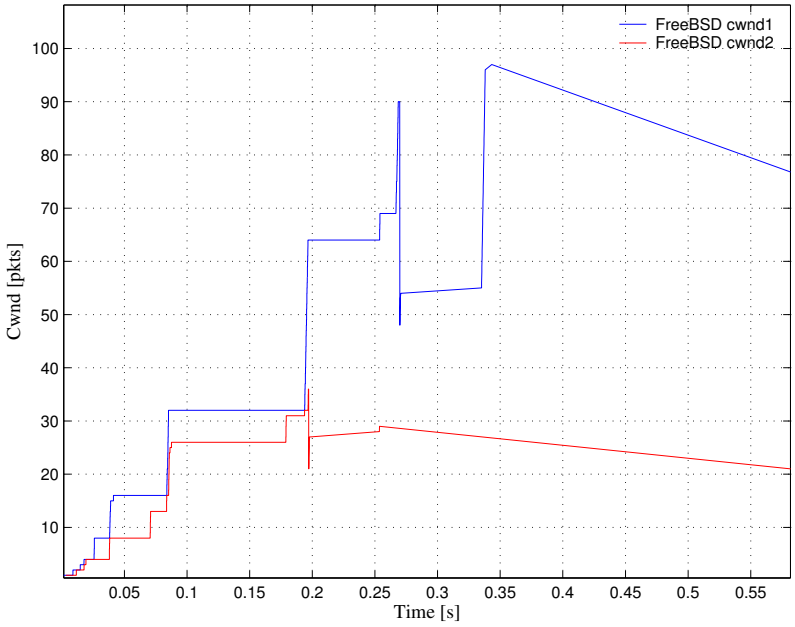

Figure 3.10: Congestion window time histories of two FreeBSD TCP flows over a live network during Slow Start

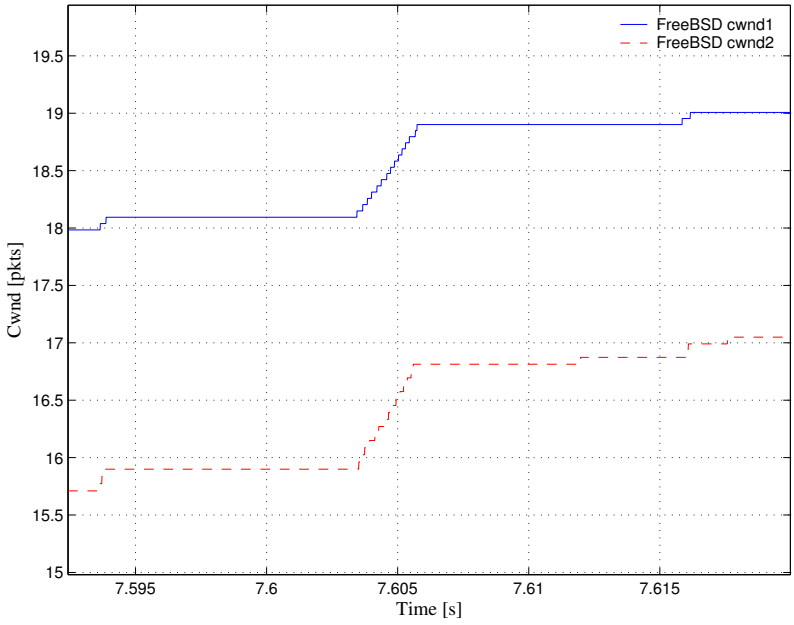

Figure 3.11: Congestion window time histories of two FreeBSD TCP flows over a live network during Congestion Avoidance

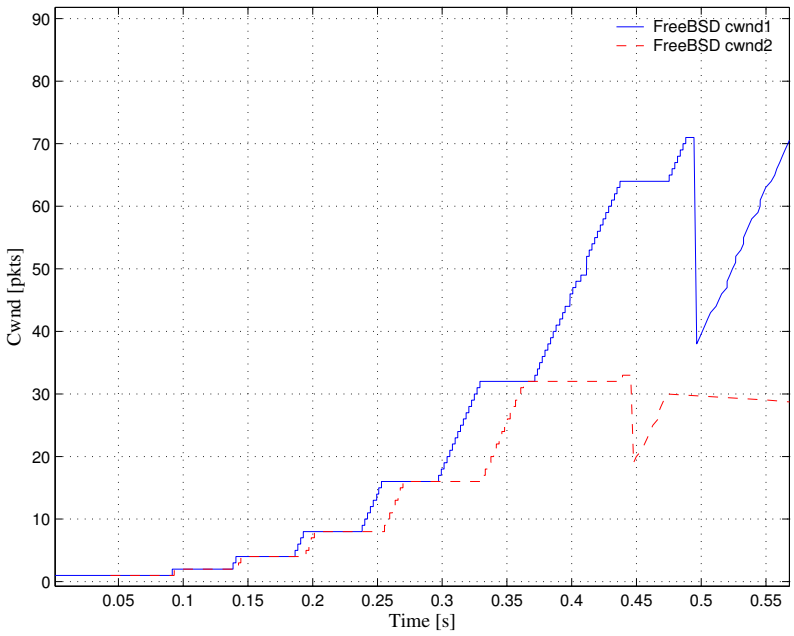

Figure 3.12: Congestion window time histories of two FreeBSD TCP flows with a 40ms delay over a live network during Slow Start

packet entrainment is present. We note that there are also cases where entrainment is somewhat broken up. Figure 3.14 shows us an example from the same experiment.

While it remains an open question as to the level of entrainment in real networks, we have seen that entrainment can indeed exist at a significant level in production networks[3].

3.3.4 Phase Effects Associated with Entrainment

One of the reasons that the prevalence of packet entrainment is of interest is that the associated phase effects are known (at least in simulation) to have a significant impact on the fair allocation of bandwidth between competing TCP flows. To explore this further, we carried out a series of experiments where the flows are run from two separate machines that are configured in the same manner. These experiments were run multiple times with different start times and ordering of which flow started first. The results were robust and consistent. Figure 3.15 shows the congestion window time histories when both flows have the same RTT. Figure 3.16 shows the same experiment but with the RTT for one flow increased by 1ms. Evidently, this small change in RTT has had a very substantial impact on the allocation of bandwidth between the flows. Similar behaviour is observed in NS,

[3]We note that there are techniques to break up entrainment such as pacing TCP[24].

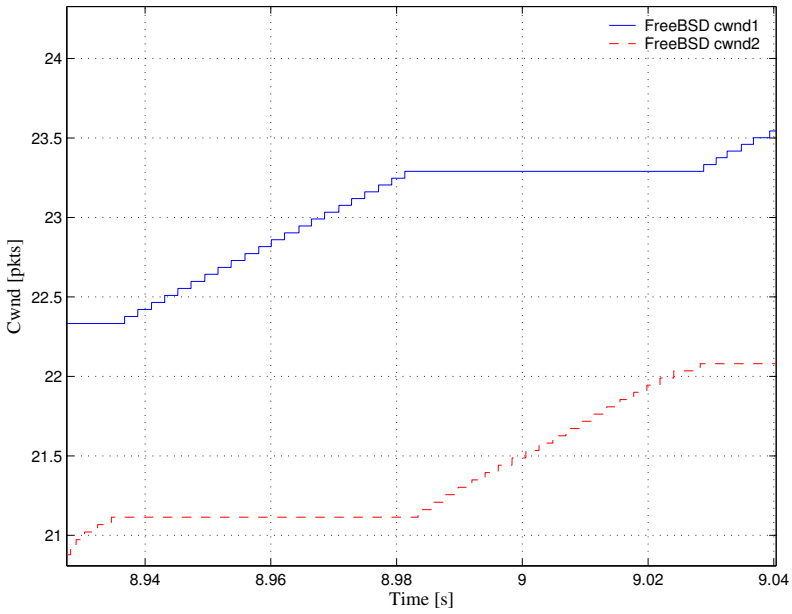

Figure 3.13: First close up comparison of two FreeBSD TCP flows with a 40ms delay over a live network during Congestion Avoidance

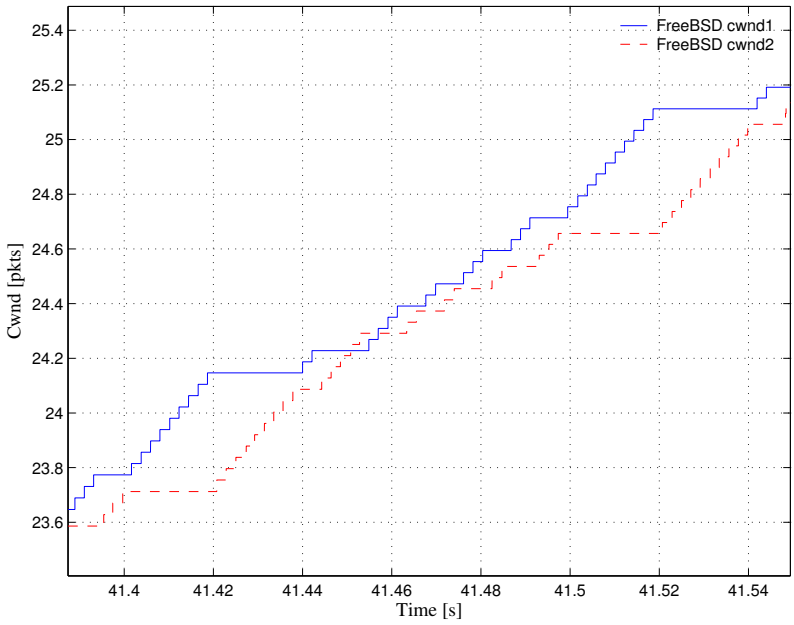

Figure 3.14: Second close up comparison of two FreeBSD TCP flows with a 40ms delay over a live network during Congestion Avoidance

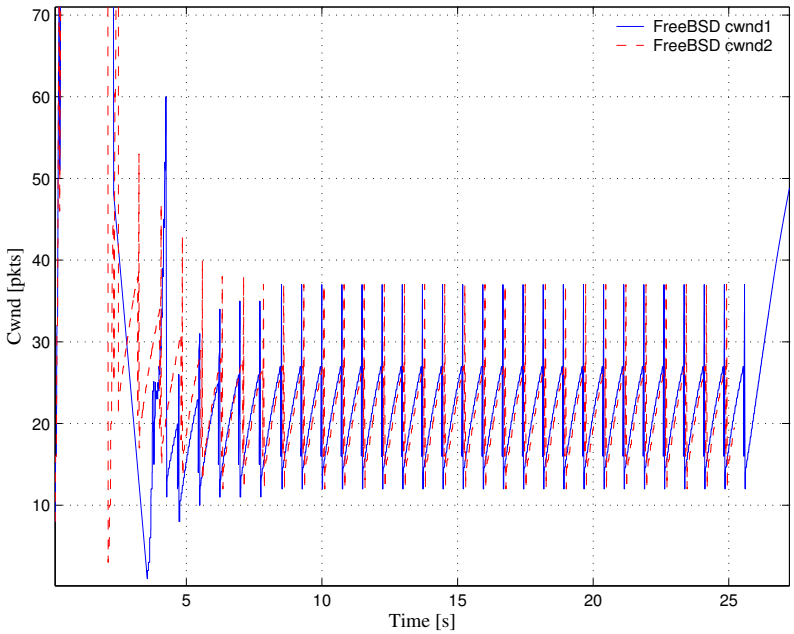

Figure 3.15: Two FreeBSD TCP flows run on separate machines with the same RTT (800Kbit/s, 100ms delay, 1000 byte packets, queue size 15 packets)

see Figure 3.17. Such effects are of course well known and have been reported for example by Floyd et al [3] but to our knowledge, this is one of the first times in open literature that phase effects have been demonstrated in a real network (as opposed to a simulation). Interestingly in our experiments we found that it was not possible to reproduce this effect under the same conditions but with the two sources running on the same machine and in the same process. Figure 3.18 shows results for two flows on the same machine with an extra 1ms in the RTT for one of the flows. The source and destination machines had separate network cards for each flow and therefore the difference is presumably due to software scheduling issues rather than ordering imposed by the hardware.

To explore this further and in particular to study the accuracy of NS predictions we repeated a number of the simulation experiments similar to those carried out by Floyd et al [3]. The network for these experiments consisted of a dumbbell network with two sources with a single flow each, one router and one destination. The bottleneck link bandwidth is 800Kbit/s, round trip propagation delay 200ms and the queue size is 15 packets. The links for the two sources have a bandwidth of 8000Kbit/s and a delay of 10ms. The packet size for the experiments is 1000 bytes. The delay on one of the source links was varied so as to change the RTT for that flow and comparisons were then made of *cwnd* for each flow. See Appendix A.3.3 for the OTcl script used in these experiments and Appendix A.3.2 for the *dummynet* configuration. Figures 3.19 and 3.21 show the NS results for 2ms and 40ms differences in RTT's between the flows. Observe the way the

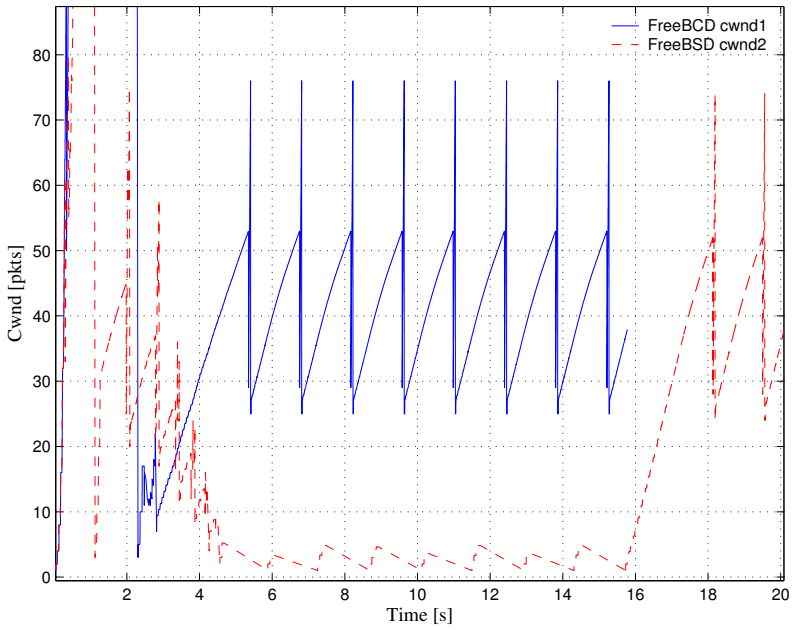

Figure 3.16: Two FreeBSD TCP flows run on separate machines with flow 1 delay 100ms and flow 2 delay 101ms (800Kbit/s, 1000 byte packets, queue size 15 packets)

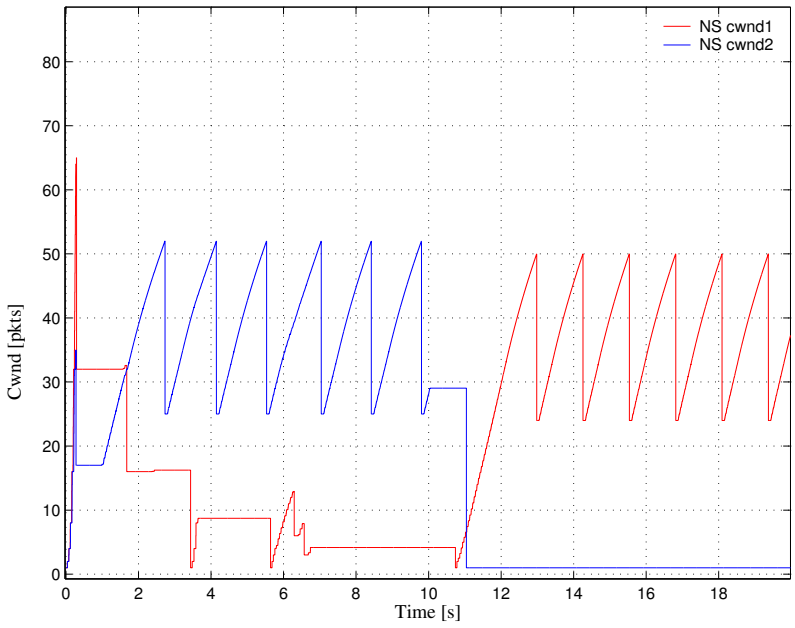

Figure 3.17: Two NS TCP flows run on separate sources with flow 1 delay 100ms and flow 2 delay 101ms (800Kbit/s, 1000 byte packets, queue size 15 packets)

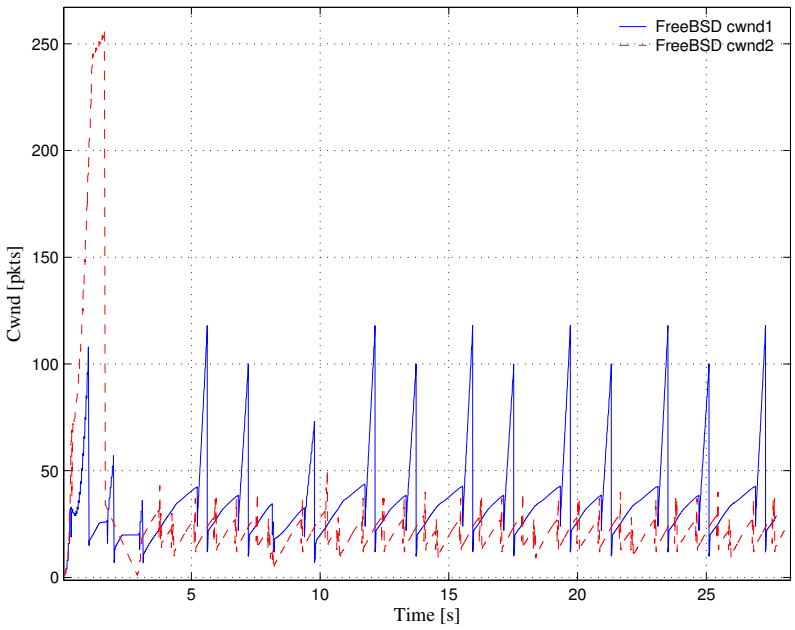

Figure 3.18: Two FreeBSD TCP flows run on the same machine with flow 1 delay 100ms and flow 2 delay 101ms (800Kbit/s, 1000 byte packets, queue size 15 packets)

unfairness between the flows flips for the two different RTT's. This is a result of the so called phase effect referred to by Floyd et al [3]. Floyd et al ran these experiments using NS only. We ran the same experiments in our test network. Figures 3.20 and 3.22 show the results. We can see similar phase effects. We note the amplitude and periodicity differences in the congestion window time histories for NS and FreeBSD.

3.3.5 Effect of Delayed ACK's

Delayed ACK's are commonly used in TCP. If this option is turned on at the destination the rate of incoming ACK's is reduced at the sender. Figure 3.23 shows what happens in our test environment when delayed ACK's are turned on. Synchronisation between flows is lost and the output rate during Slow Start and Congestion Avoidance is reduced, see Figures 3.24 and 3.25. Figure 3.26 shows the results for the same experiment run in NS. The congestion window time histories are significantly different than the results from the test network. Close ups of Slow Start, Figure 3.27, and Congestion Avoidance, Figure 3.28, also show significant differences for congestion window time histories when compared to Figures 3.24 and 3.25.

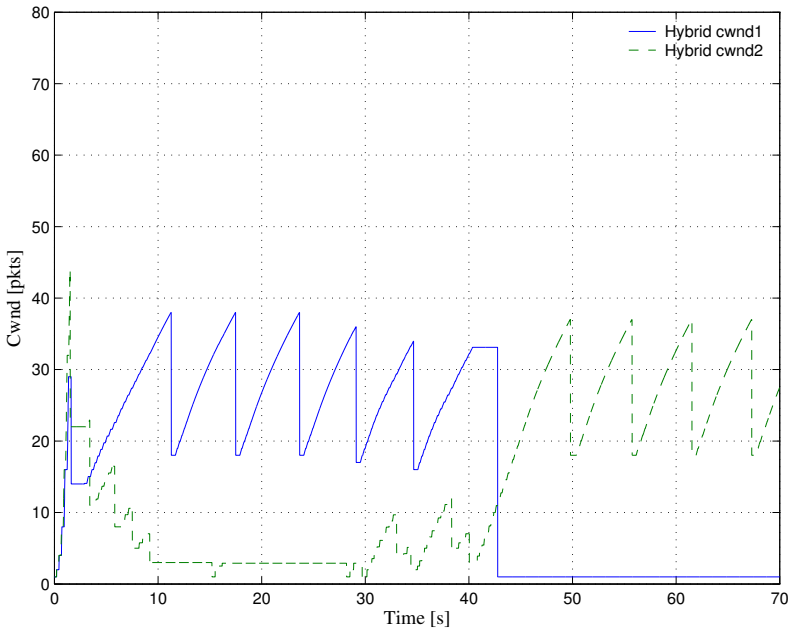

Figure 3.19: Two NS TCP flows run on the different machines with flow 1 delay 210ms and flow 2 delay 208ms (800Kbit/s, 1000 byte packets, queue size 15 packets)

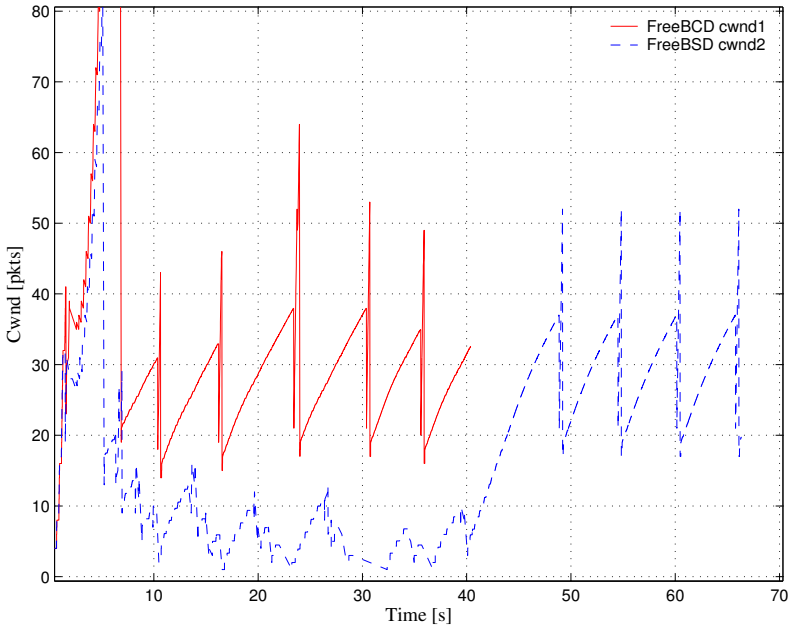

Figure 3.20: Two FreeBSD TCP flows run on the different machines with flow 1 delay 210ms and flow 2 delay 208ms (800Kbit/s, 1000 byte packets, queue size 15 packets)

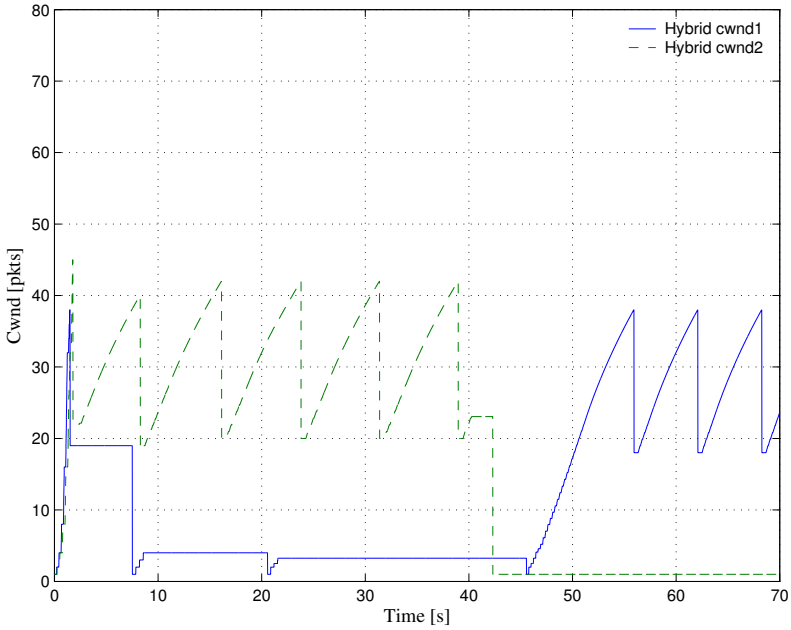

Figure 3.21: Two NS TCP flows run on different sources with flow 1 delay 210ms and flow 2 delay 250ms (800Kbit/s, 1000 byte packets, queue size 15 packets)

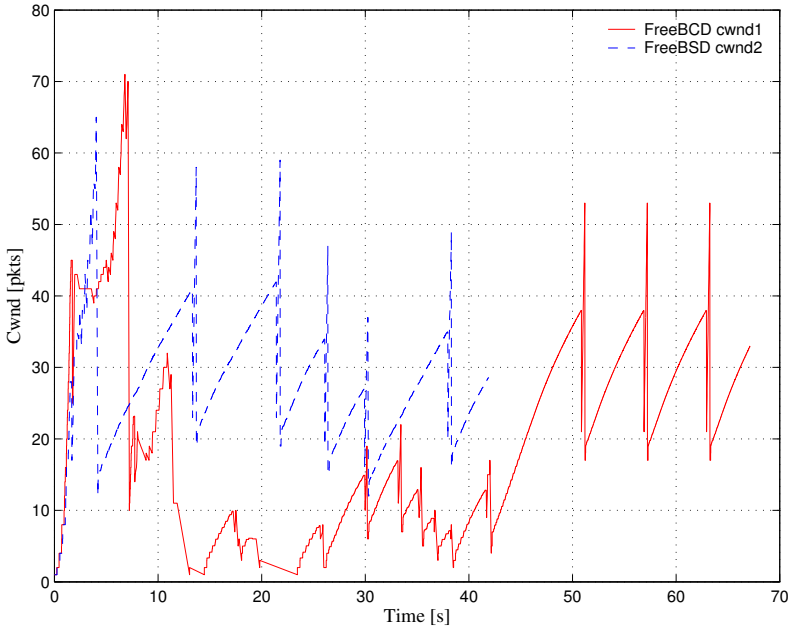

Figure 3.22: Two FreeBSD TCP flows run on the different machines with flow 1 delay 210ms and flow 2 delay 250ms (800Kbit/s, 1000 byte packets, queue size 15 packets)

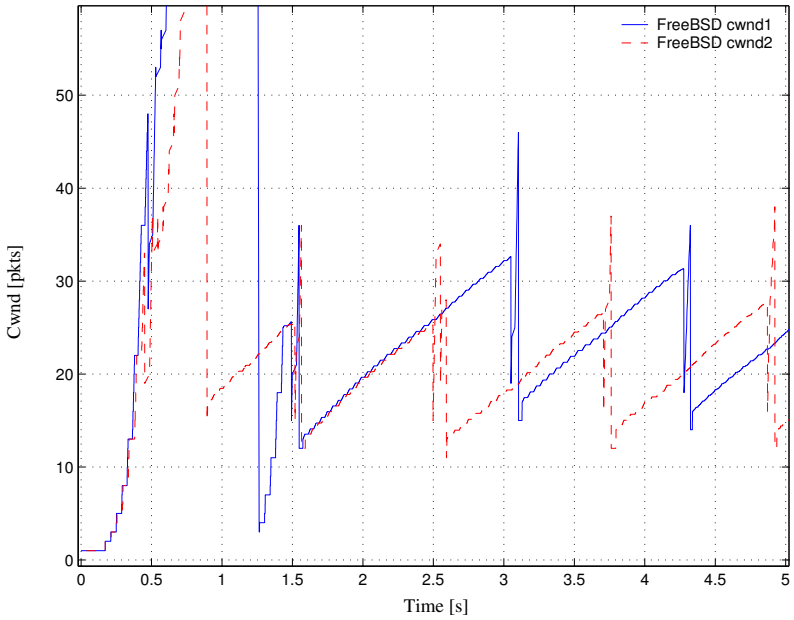

Figure 3.23: Congestion window time histories of two FreeBSD TCP flows with delayed ACK's (10Mbit/s, 40ms delay, queue size 17 packets, delayed ACK interval 100ms)

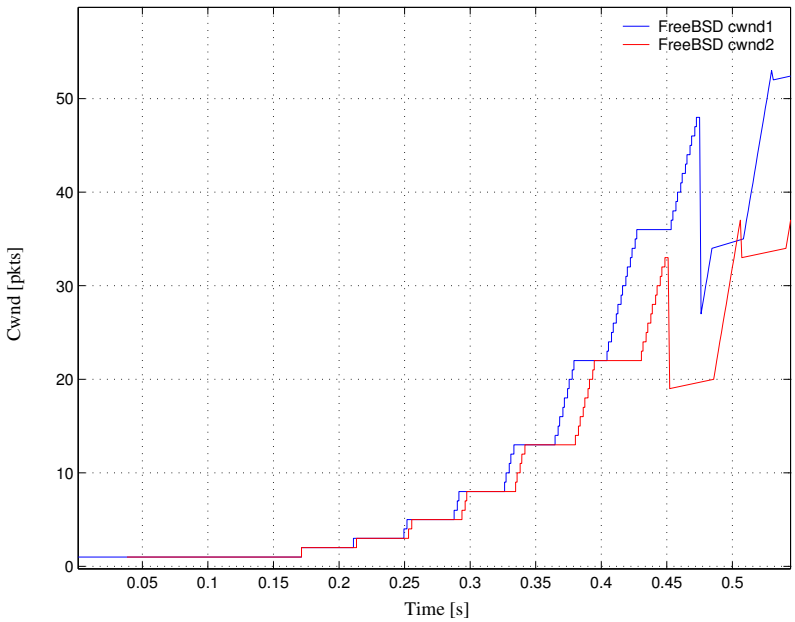

Figure 3.24: Close up of comparison of two FreeBSD TCP flows with delayed ACK's during Slow Start (10Mbit/s, 40ms delay, queue size 17 packets, delayed ACK interval 100ms)

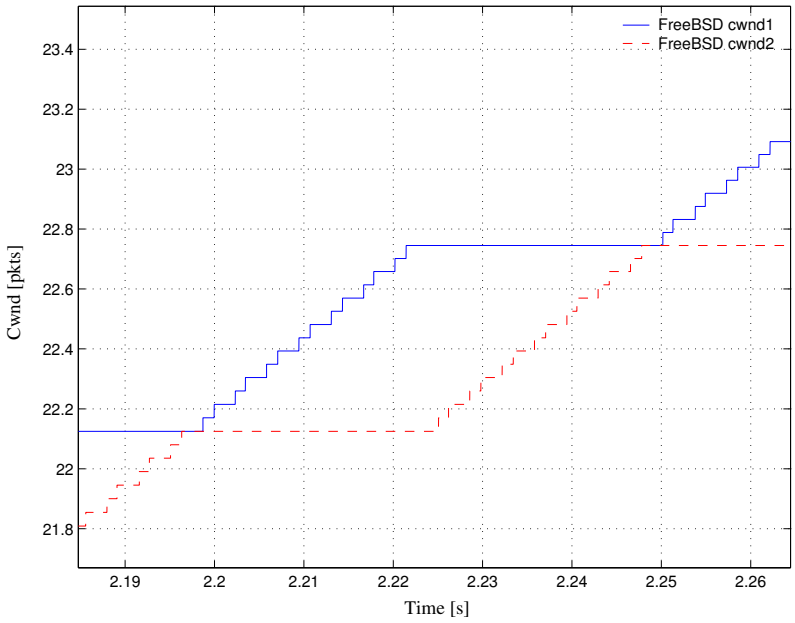

Figure 3.25: Close up of comparison of two FreeBSD TCP flows with delayed ACK's during Congestion Avoidance (10Mbit/s, 40ms delay, queue size 17 packets, delayed ACK interval 100ms)

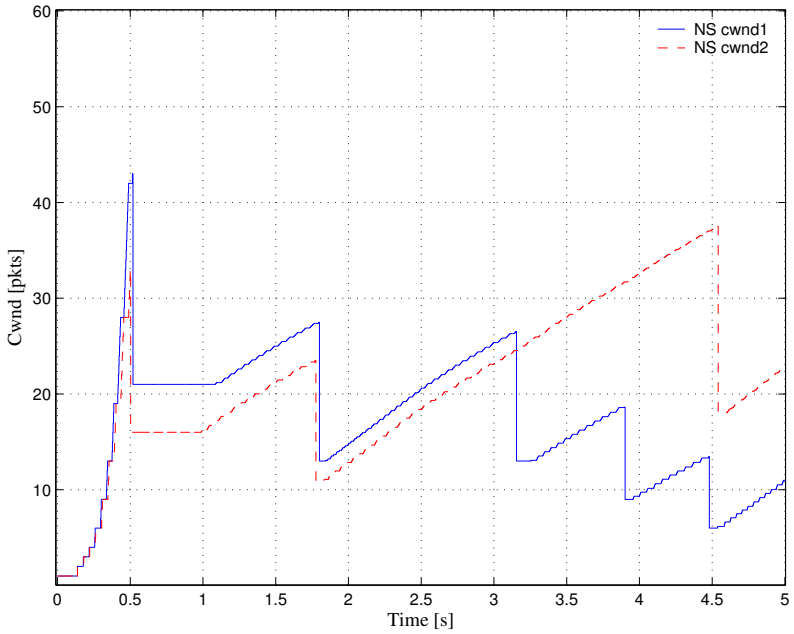

Figure 3.26: Congestion window time histories of two NS TCP flows with delayed ACK's (10Mbit/s, 40ms delay, queue size 17 packets, delayed ACK interval 100ms)

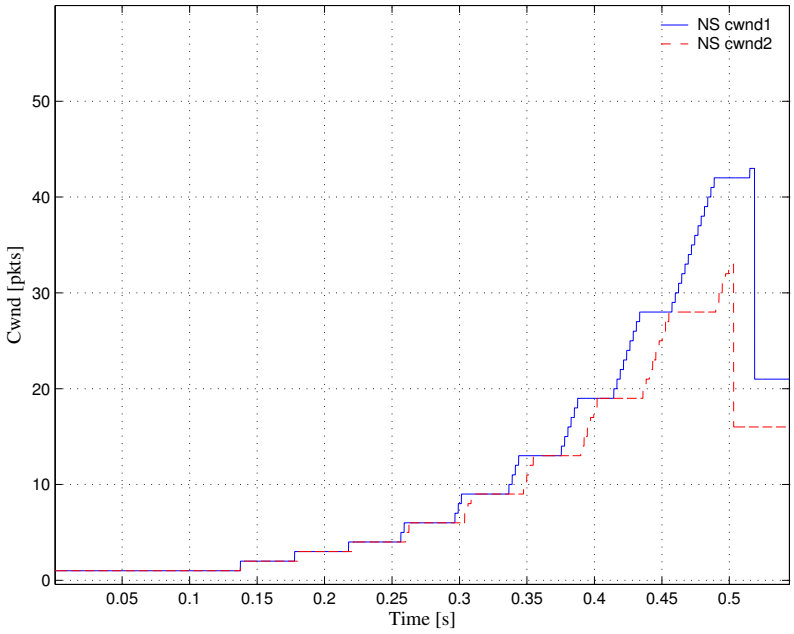

Figure 3.27: Close up of comparison of two NS TCP flows with delayed ACK's during Slow Start (10Mbit/s, 40ms delay, queue size 17 packets, delay interval 100ms)

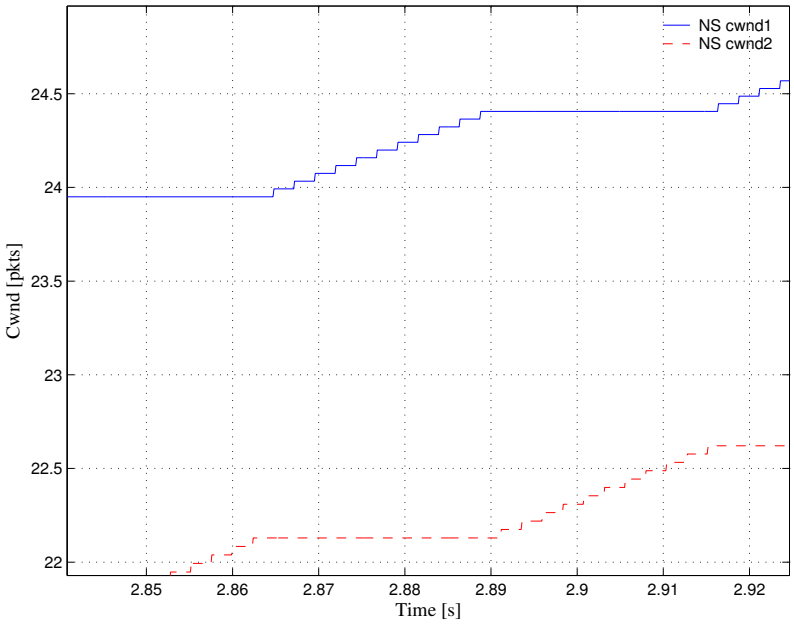

Figure 3.28: Close up of comparison of two NS TCP flows with delayed ACK's during Congestion Avoidance (10Mbit/s, 40ms delay, queue size 17 packets, delay interval 100ms)

3.4 Summary

In this chapter we developed a test network and instrumented the FreeBSD TCP stack. We compared NS to the testbed for one and two flows in both Slow Start and Congestion Avoidance. We demonstrated the presence of entrainment in a live network. We also confirmed phase effects in our test network. The results above point to entrainment effects being a determining factor in flow fairness under certain network conditions. Indeed in certain circumstances lockouts can occur. This was a pilot study and more work is clearly required to validate NS properly. However on the basis of our tests, NS was found to be remarkably accurate for the synchronised flow case. For the non synchronised flow and delayed ACK cases NS was found to be less accurate.

Chapter 4

Comparison of Hybrid and NS Models

Our focus in this chapter is on bottleneck networks employing a drop-tail queueing policy. While models have been proposed with active queueing such as RED [25], far fewer exist for drop-tail queuing. One model that supports drop-tail queuing is the hybrid model for TCP proposed by Bohacek et al [4]. Hybrid dynamic systems involve a mixture of continuous/discrete dynamics and logic based switching [48, 49, 50]. Typically, these systems evolve according to underlying continuous/discrete dynamics and experience abrupt mode changes triggered by a set of conditions within the system. In this chapter we compare the hybrid model for TCP proposed by Bohacek et al [4] with NS. The hybrid model includes TCP mode transitions, the impact of time varying delays on *cwnd* evolution and drop-tail queuing while working within a fluid-like framework.

The chapter is organised as follows. Section 4.1 presents the details of the hybrid model. In section 4.2 we compare the hybrid model with NS. In section 4.3 we investigate modifying the hybrid model to include a discrete queue as a means of developing a more powerful model of packet drop taking account of the effect of discrete events inherent in TCP on the hybrid model. In section 4.4 we provide a summary of our findings.

4.1 Hybrid Model

The hybrid model splits the network into TCP sources and network queues. We consider each in turn.

4.1.1 Source Model

The main modes for a TCP source under congestion control are Slow Start, Fast Recovery, Fast Retransmit and Timeout. See Figure 4.1.

Slow Start:

TCP starts in Slow Start mode where the congestion window w_f for flow f doubles every round trip time. This exponential growth is modelled as follows

$$\dot{w_f} = \frac{\log m}{RTT_f} w_f \tag{4.1}$$

where m is a constant that dictates the growth rate and RTT_f is the RTT for flow f. Assuming doubling of w_f every RTT_f and constant RTT_f the value of m is 2. The instantaneous per flow send rate r_f is given by

$$r_f = \frac{\beta w_f}{RTT_f} \tag{4.2}$$

The constant β is introduced by Bohacek et al, to capture effects associated with the rapid change in RTT observed in Slow Start [4]. From empirical evaluation of NS traces, Bohacek et al determined a value of $\beta = 1.45$ as a suitable corrective factor.

The TCP source exits Slow Start if w_f is greater than the Slow Start threshold, $ssthr_f$, otherwise w_f increases until a packet is dropped at the queue. There is a delay between when a packet is dropped and when the TCP source detects the drop. To model this as a drop the TCP source moves into a Slow Start delay mode where it continues to increase w_f and r_f as per (4.1) and (4.2) for the duration of the drop detection delay DDD_F, The source then moves to either Fast Recovery mode or Timeout mode depending on the number of packets lost, n_{drop}. If $n_{drop} \geq w_f - 2$ there will not be enough duplicate ACK's to trigger a transition to Fast Recovery mode and a timeout will result. If $n_{drop} \geq \frac{w_f}{2} + 2$ there will not be enough ACK's to allow the retransmission of all the packets that were dropped and a timeout will again result.

Fast Recovery:

In Fast Recovery w_f and r_f are reset to

$$w_f = \frac{w_f^-}{2}, \quad r_f = \frac{1 + w_f^-/2 - n_{drops}}{RTT_f} \tag{4.3}$$

where w_f^- is the value of w_f at the time when the first packet drop is detected by the

source. The number of RTT_f's the source should remain in Fast Recovery is given by

$$k = \lceil \log_2 \frac{1+w_f^-/2}{1+w_f^-/2-n_{drops}} \rceil \tag{4.4}$$

This is only true where $n_{drops} \leq w_f^-/2 + 1$ otherwise the number of RTT_f's the TCP source remains in Fast Recovery is given by

$$k = 1 + \lceil \log_2 n_{drops} \rceil \tag{4.5}$$

Congestion Avoidance:

In Congestion Avoidance mode w_f increases linearly at a rate of one packet per RTT_f. This is modelled by

$$\dot{w_f} = \frac{1}{RTT_f} w_f \tag{4.6}$$

The instantaneous send rate r_f is

$$r_f = \frac{w_f}{RTT_f} \tag{4.7}$$

The TCP flow remains in Congestion Avoidance until a packet drop occurs. There is a delay between when a packet is dropped and when the TCP source detects the drop. To model this the TCP source moves into a Congestion Avoidance delay mode which behaves in a similar manner to the Slow Start delay mode. After delay DDD_F, the source moves to either Fast Recovery mode or Timeout mode.

Timeout:

The Timeout mode is entered when

$$w_f \leq max\{2 + n_{drop}, 2n_{drop} - 4\} \tag{4.8}$$

and the transition to the Timeout mode is only possible from either Slow Start delay mode or Congestion Avoidance delay mode. The model assumes that the TCP source remains in this mode for 1 second and that $r_f = 0$. After this period w_f is reset to one, $ssthr_f$ is reset to $\frac{w_f^-}{2}$ where w_f^- is the value of w_f at the time when the first packet drop

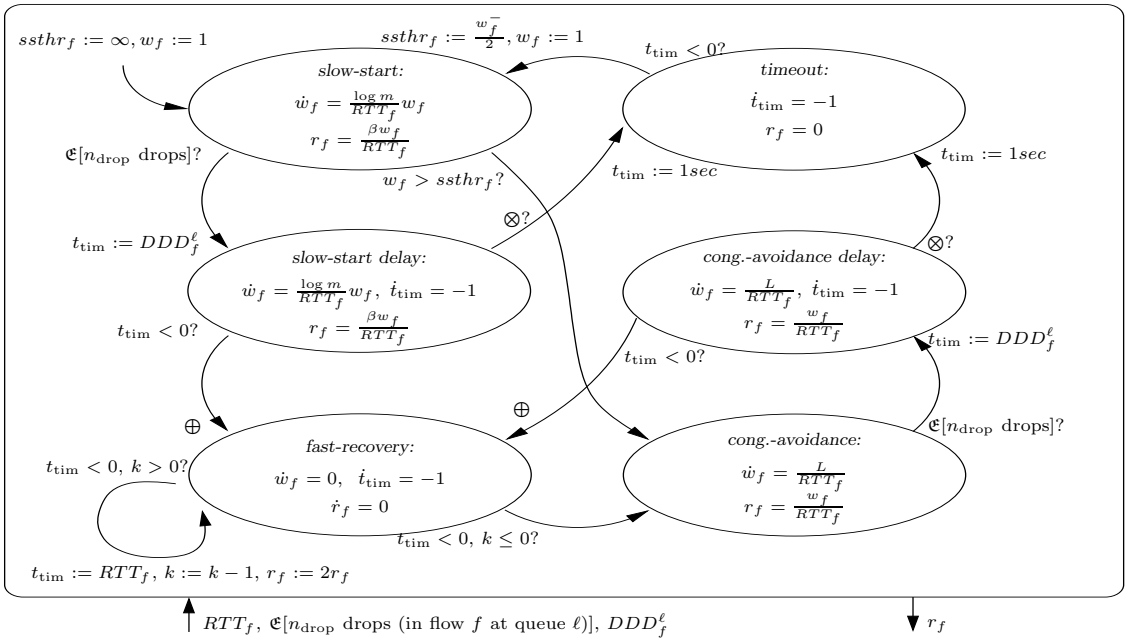

Figure 4.1: Hybrid model for flow f using TCP-Sack congestion control. The symbol $\bigotimes$ represents $w_f \leq max\{2+n_{drop}, 2n_{drop}-4\}$ and $\bigoplus$ represents $t_{tim} = RTT_f, w_f = \frac{w_f^-}{2}, r_f = \frac{1+w_f^-/2-n_{drop}}{RTT_f}$ and k given by (4.4) and (4.5)[1]

is detected by the source.

4.1.2 Queue Model

Queue Empty:

Similarly to the sources, a fluid queue model is proposed by Bohacek et al [4] which involves three modes: Queue Empty, Queue Not Full and Queue Full. Let F be the set of flows $\{f_1, f_2, ...f_n\}$ and L be the set of links $\{l_1, l_2, ...l_n\}$. In the Queue Empty mode let s_f^l denote the input rate for flow f on link l and let r_f^l denote the output rate for flow f on link l. The queue size is modelled as a first order differential equation given by

$$\dot{q_f^l} = s_f^l - r_f^l \qquad \forall f \epsilon F, l \epsilon L \tag{4.9}$$

The output rate r_f^l of the queue equals the input rate when the total input rate s_l is less

[1]This diagram is reproduced from work by Bohacek et al [4].

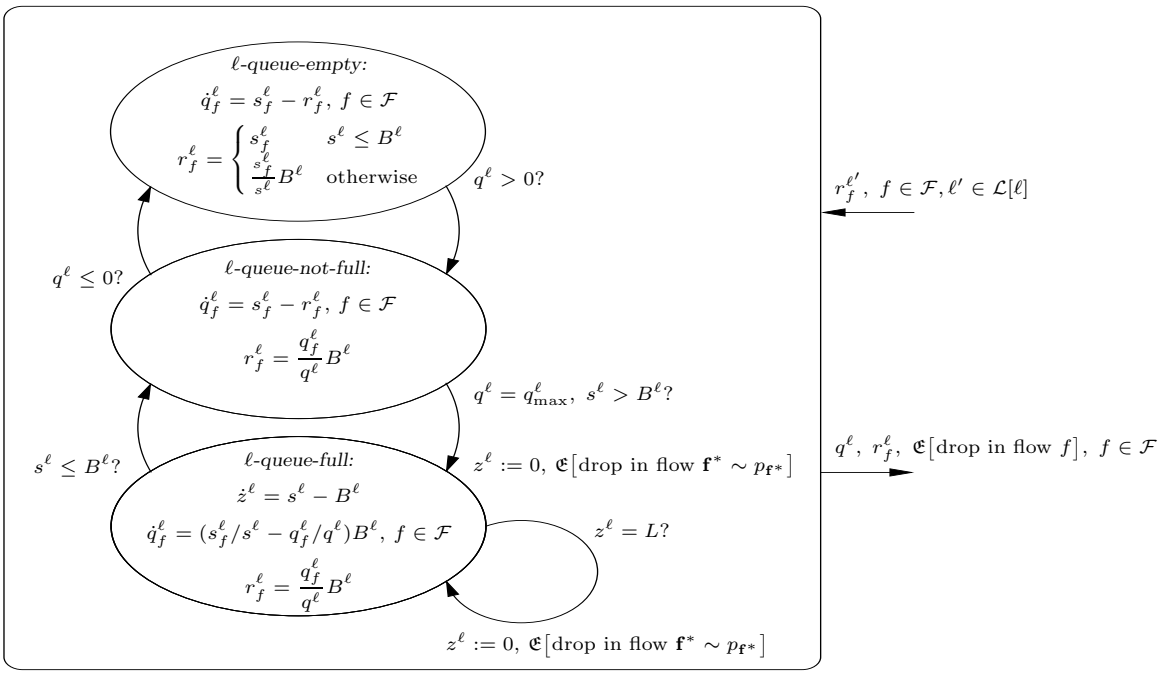

Figure 4.2: Hybrid model of TCP Network Queue at link l where $q_l = \sum_{f \epsilon F} q_l^f$ and $s_l = \sum_{f \epsilon F} s_l^f, \forall l \epsilon L$[2]

than or equal to the queue service rate B^l. Otherwise the output rate is B^l is distributed between the incoming flows based on the proportion each contributes to the total input s^l. That is

$$r_f^l = \left\{ \begin{array}{ll} s_f^l & s^l \le B^l \\ \frac{s_f^l}{s^l} B^l & otherwise \end{array} \right\} \tag{4.10}$$

Queue Not Full:

In the queue not full mode the queue size model is as in (4.9) but the output rate r_f^l is distributed among the flows based on the proportion of the queue they occupy so that

$$r_f^l = \frac{q_f^l}{q_l} B^l \tag{4.11}$$

[2]This diagram is reproduced from work by Bohacek et al [4].

where q_l is the total size of the queue.

Queue Full:

In the Queue Full mode the output rate is modelled by (4.11) and the queue size is given by

$$\dot{q_f^l} = (\frac{s_f^l}{s_l} - \frac{q_f^l}{q_l})B^l \qquad \forall f \epsilon F, l \epsilon L \tag{4.12}$$

The excess fluid z_l arriving at the queue is given by

$$\dot{z^l} = s^l - B^l \tag{4.13}$$

This aggregated fluid represents the total volume of dropped packets and must be assigned to the input flows. This is a key issue in accurately modelling the behaviour of TCP. When drops are synchronised (every flow sees a drop when the queue is full), Bohacek et al considers a drop rotation model whereby dropped packets are assigned to flows in a round-robin fashion. Bohacek et al also consider assigning drops randomly between flows using a uniform distribution.

4.1.3 Complete Model

The queue and TCP source model must be combined to obtain a complete network model. The round-trip time RTT_f, the drop detection delay DDD_f and the number of drops n_{drop} are inputs to the TCP source model. The send rate r_f^l is the queue model input. The complete model for the dumbbell topology has been implemented in Matlab (see Appendix A.4.1)

4.2 Comparison of Hybrid Model with NS

The hybrid model involves many approximations that must to be validated. As a first step we look at a single flow on a dumbbell network topology. Figure 4.3 shows a comparison of the NS model and the hybrid model for a single flow running on a network with a 10Mb link, 40ms delay, queue length of 17 packets and packet size of 1500 bytes. We note that NS models the transmission delay over the link and the hybrid model omits this delay so we have reduced the fixed delay in the NS simulations by $1/B$, where B is the bandwidth

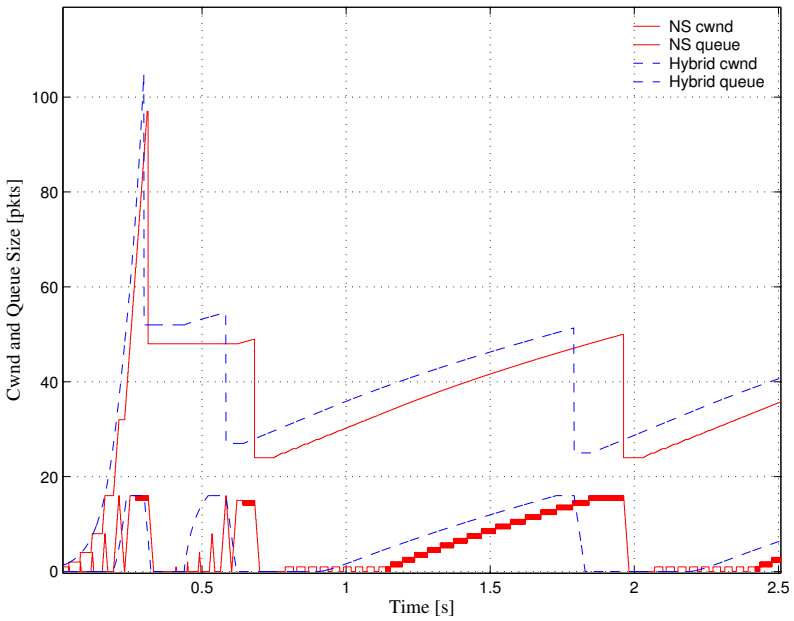

Figure 4.3: Comparison of a single flow for NS and hybrid models (10Mbit/s, 40ms delay, queue size 17 packets)

of the bottleneck link, to compensate. We only consider the Slow Start and Congestion Avoidance modes here as the Fast Recovery mode and the Timeout mode are dependent only on the packets dropped in the previous modes.

Slow Start:

As can be seen, our comparison with NS highlights some issues with the model proposed by Bohacek et al [4]. We initially look at Slow Start. Figure 4.4 compares the output from the hybrid model and NS during Slow Start. The entrainment effect can clearly be seen in the NS plot as a rapid increase in *cwnd* followed by a dead-time. This effect is due to the back-to-back transmissions of packets from the source coupled with inter packet delay introduced by the $1/B$ delay due to packet transmission time over the link.

The total network capacity (queue plus link delay-bandwidth product) for a TCP flow before a drop is 50 packets. In the NS model the first drop occurs at 0.2511s when *cwnd* is 48 packets. This is probably due to the entrainment of packets causing the queue to overflow earlier as the effective input rate is greater that the averaged input rate on which the total network capacity calculation of 50 packets is based. The hybrid model drops a packet at a w_f of 52.48.

A second difference between the hybrid and NS model predictions can also be seen in Figure 4.4. The NS *cwnd* time history plot is laterally displaced compared to the time

history predicted by the hybrid model. This is thought to be due to the entrained packets in NS causing increases in the queue length as received ACK's generate back-to-back pairs of packets from the source which causes the second packet to be queued as the first is transmitted. These queue increases are cumulative and increase exponentially as packets sent onto the network are acknowledged and additional packets are sent out back-to-back. This leads to the triangular patterns seen in Figure 4.5 which have peaks of $cwnd/2$ (every second packet is queued) and occur once per RTT. These queue increases cause increases in RTT as

$$RTT = RTT_f + q_l/B \tag{4.14}$$

These packet level effects are not modelled by the fluid-like queue in the hybrid model. The difference at the Slow Start peak of the NS plot is due to the sum of these additional delays[3] which is 1.2ms + 2.4ms + 4.8ms + 9.6ms which is 18ms. This is the delay between the NS model and the hybrid model and can be seen in Figure 4.4 when *cwnd* reaches 97 packets for the NS plot. We do not include the delay due to the triangular peak starting at 19ms as the hybrid model queue captures this delay.

An important feature for Fast Recovery and Timeout is the drop model used and specifically the number of dropped packets. This affects the recovery time for the source and thus affects the time the models spend in Fast Recovery mode. In the NS model 49 packets are dropped and in the hybrid model 44 packets are dropped. As can be seen in Figure 4.3, NS spends longer in Fast Recovery mode compared to the hybrid model. This is again thought to be an artifact of the entrainment effect as the entrained input rate to the queue will be higher than the averaged input rate seen by the hybrid model queue and thus the NS queue will see more packet drops.

Congestion Avoidance:

The differences in Slow Start outlined above affect the initial conditions for Congestion Avoidance in the hybrid model. We therefore artificially force the initial conditions of the hybrid model in Congestion Avoidance to match those observed in NS. The entrainment issues outlined above also affect Congestion Avoidance mode but are less pronounced as the rate of increase of *cwnd* is less than in Slow Start mode. Back-to-back packet transmission will only occur when an ACK is received that causes *cwnd* to pass through an integer value and thus only once per RTT. This means that we will only see one dropped packet per flow per congestion epoch and $1/B$ extra delay per epoch (caused by the second of the back-to-back packets remaining in the queue while the first packet is transmitted) . Figure 4.6 shows a comparison for single flows starting in Congestion Avoidance. The results are quite similar with the main differences being the overshoot

[3]Line rate is 1/B which is 1.2ms for a 10Mb link with a packet size of 1500 bytes.

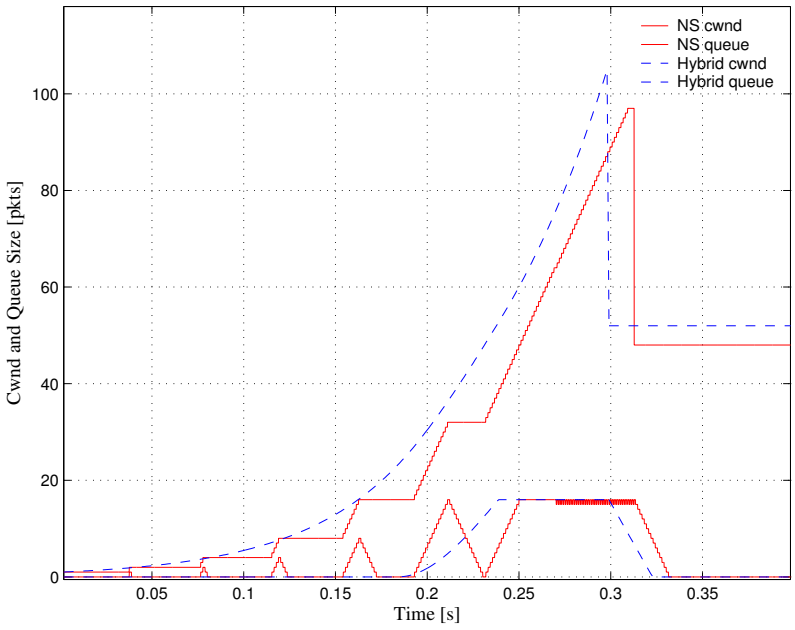

Figure 4.4: Comparison of NS and hybrid model congestion window during Slow Start (10Mbit/s, 40ms delay, queue size 17 packets)

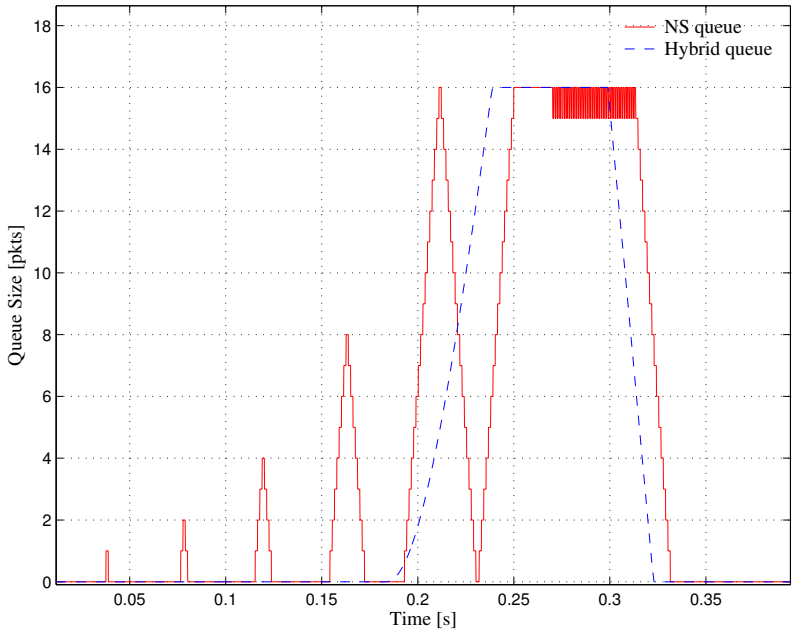

Figure 4.5: Comparison of NS and hybrid model queues during Slow Start (10Mbit/s, 40ms delay, queue size 17 packets)

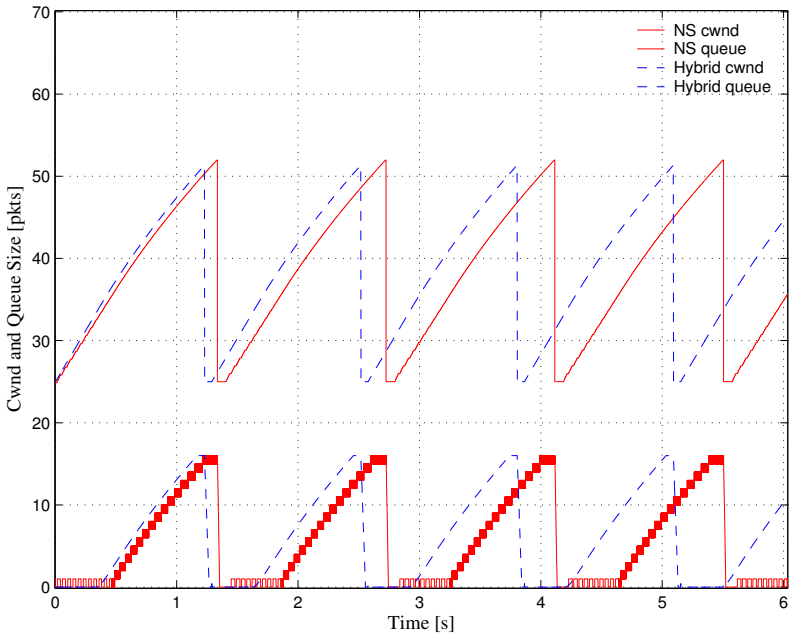

Figure 4.6: Comparison of NS and hybrid model during Congestion Avoidance (10Mbit/s, 40ms delay, queue size 17 packets)

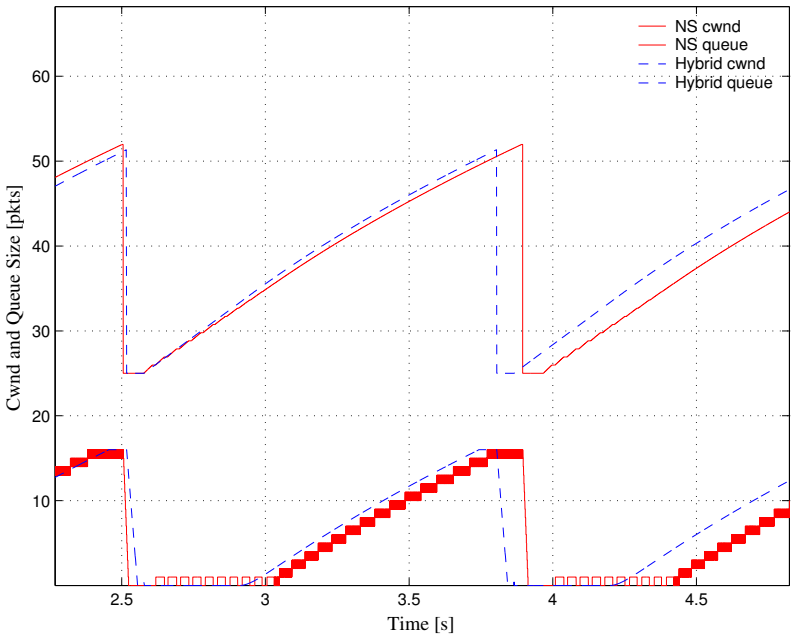

Figure 4.7: Close up comparison of NS and hybrid model during Congestion Avoidance (10Mbit/s, 40ms delay, queue size 17 packets)

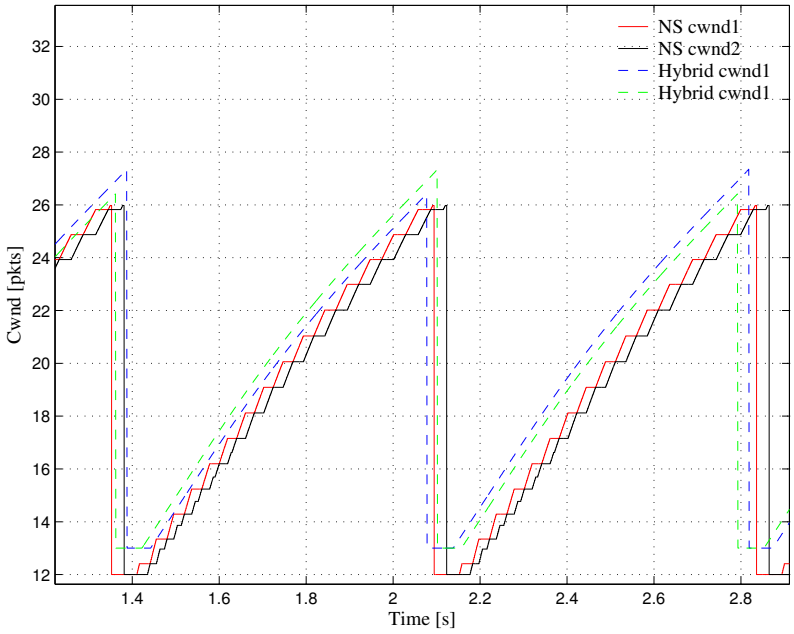

Figure 4.8: Comparison of NS and hybrid model with two flows for Congestion Avoidance (10Mbit/s, 40ms delay, queue size 17 packets)

by the hybrid model. Figure 4.7 shows a more detailed view of part of the plot in Figure 4.6. We can see a difference in the slope of *cwnd* between the two models and a difference in the Fast Recovery length. Figure 4.8 shows the corresponding comparison for two flows. We see similar behaviour to that seen in the single flow case. Note the difference in *cwnd* peaks between the models is highlighted here as the hybrid model crosses the integer boundary at 26 packets whereas the NS model does not. This leads to different starting points for the flows in each model following Fast Recovery. We note also the small difference in slope. The Fast Recovery times seen in both models are similar.

4.3 Hybrid Model with Discrete Queue

A key aspect of the hybrid model is the way in which packet drops are modelled. While round-robin dropping is evidently accurate when drops are synchronised the formulation of a drop model under more general conditions is an open problem. The fundamental problem here is that packet drops are discrete in nature (the queue does not drop fractions of packets). It seems natural therefore to consider using a discrete rather than continuous model of the queue. Retaining the hybrid source model of Bohacek et al [4] we consider modelling the queue as a discrete FIFO buffer as follows.

4.3.1 Discrete Queue Model

Let Q denote an ordered sequence with elements $Q_i \epsilon \{1, 2...n_f\}, i \epsilon [1, q_{max}]$ and n_f is the number of flows. Let q denote the number of elements in Q. Let $Q \bigoplus f$ denote the right addition operator such that $(Q \bigoplus f)_i = Q_i, i \epsilon [1, q]$ and $(Q \bigoplus f)_{q+1} = f$. Let σ denote the left shift operator such that $(\sigma Q)_i = Q_{i+1}$. On arrival at the queue of a packet from flow f we have

$$Q \leftarrow \left\{ \begin{array}{cc} Q \bigoplus f & q < q_{max} \\ Q & otherwise \end{array} \right\} \tag{4.15}$$

and

$$n_{dropf} \leftarrow \left\{ \begin{array}{cc} n_{dropf} & q < q_{max} \\ n_{dropf} + 1 & otherwise \end{array} \right\} \tag{4.16}$$

and when the queue is serviced

$$Q \leftarrow \sigma Q \tag{4.17}$$

Packet arrivals are calculated by simply integrating incoming fluid and marking a packet arrival when p_f^l crosses an integer boundary.

$$\dot{p_f^l} = s_f^l \tag{4.18}$$

Inputs to the queue model are therefore the send rates from each flow . Outputs from the queue are the number of drops per flow n_{dropf} and RTT_f. The automaton for this model is shown in Figure 4.9.

The send rate r_f of source f is given by $\frac{w_f}{RTT_f}$ on the assumption that w_f packets are in flight. Packet drops change the number of packets in flight so in addition to the discrete queue model the input to the queue is modified to

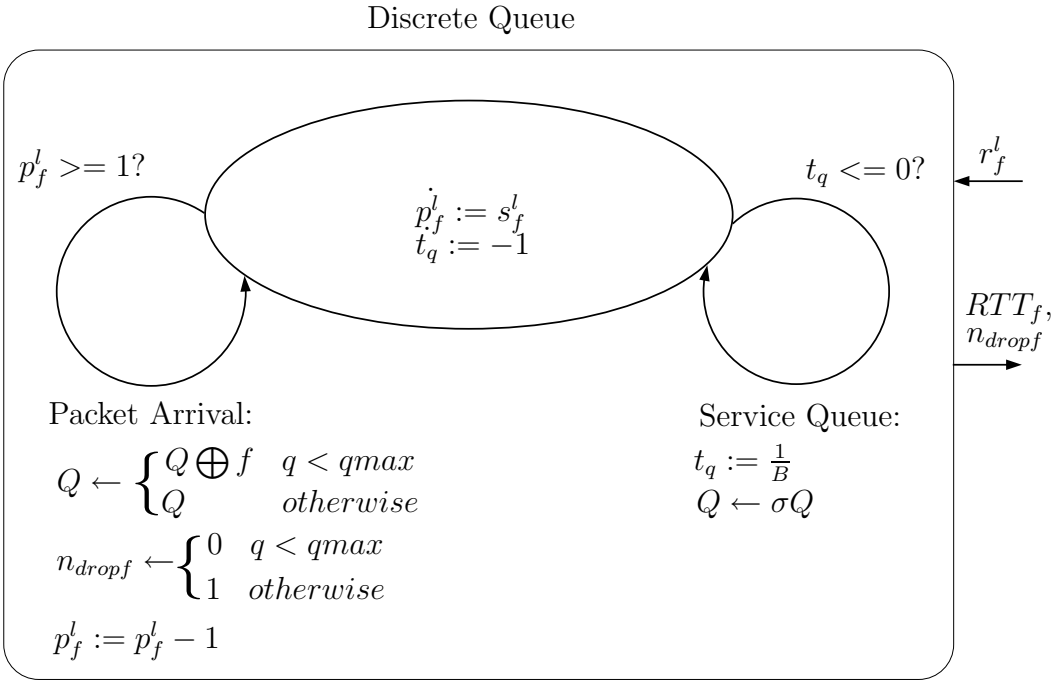

Figure 4.9: Discrete queue for Hybrid model

$$r_f = \frac{w_f}{RTT_f} - \frac{n_{dropf}}{RTT_f} - \frac{n_{dropf}}{w_f^+ RTT_f} \tag{4.19}$$

where n_{dropf}/RTT_f accounts for the discrepancy between w_f and the actual number of packets in flight and $n_{dropf}/w_f^+ RTT_f$ accounts for the corresponding change in the number of ACK's with w_f^+ denoting w_f at the time when the drop is detected at the source.

The Matlab code to implement this discrete model is given in Appendix A.4.2.

Figure 4.10 shows a comparison between NS and the hybrid model with discrete queue. It can be seen that similar results are obtained as with the standard hybrid model with a continuous queue for a single flow (compare with Figure 4.3) but with the capability to examine discrete effects in the queue.

Congestion window time histories for the hybrid model with discrete queue model for two flows can be seen in Figure 4.11. The results for the same experiment in NS can be seen in Figure 4.12. In Slow Start mode for the hybrid model with discrete queue, more dropped packet are assigned to flow 2 (25 packets) rather than flow 1 (15 packets). This causes flow 1 to enter Timeout mode while flow 2 enters Fast Recovery mode. In addition we

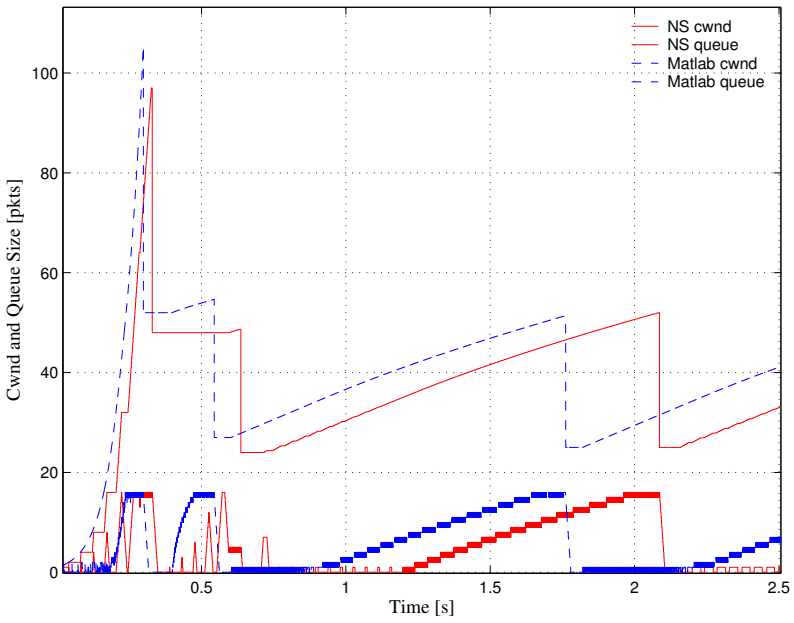

Figure 4.10: Comparison of NS and hybrid model with discrete queue for a single flow (10Mbit/s, 40ms delay, queue size 17 packets)

can see that synchronisation between the flows is lost in Congestion Avoidance mode.

4.3.2 Discrete Queue with Entrained Packets

We implemented a modification to the discrete queue model to investigate the mechanism leading to drop synchronisation. We implemented entrainment of packets at the queue concentrating on Slow Start and Congestion Avoidance modes.

Slow Start

The entrainment for Slow Start is implemented by sending trains of packets to the queue as w_f cross integer boundaries of powers of two. We pass a back-to-back packet pair into the queue, delay by the line rate and entrain the next back-to-back pair. Thus one packet is serviced and one packet is queued. We send a train of packets to the queue when

$$w_f > 2^n, state_f = SS \tag{4.20}$$

where $state_f$ is the current source mode for flow f and n is a counter initially set to one

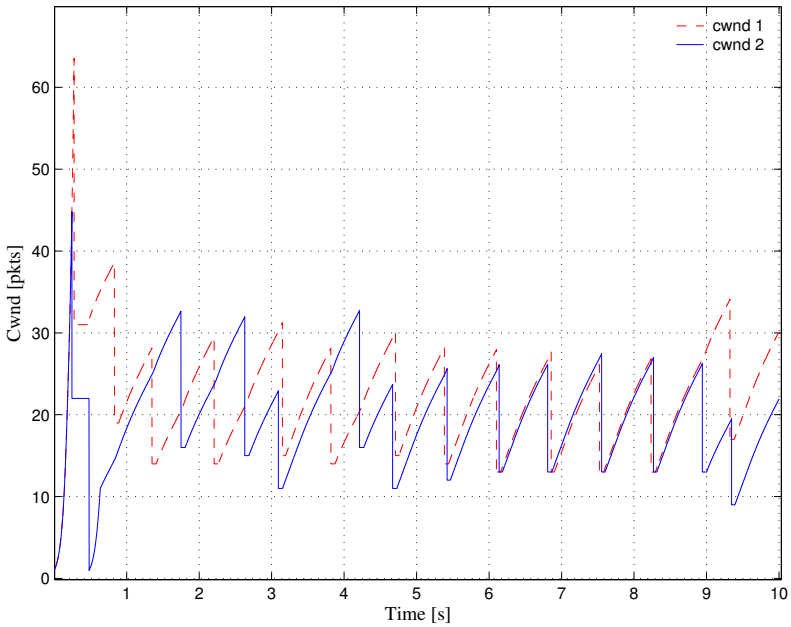

Figure 4.11: Two TCP flows from the hybrid model with discrete queue (10Mbit/s, 40ms delay, queue size 17 packets)

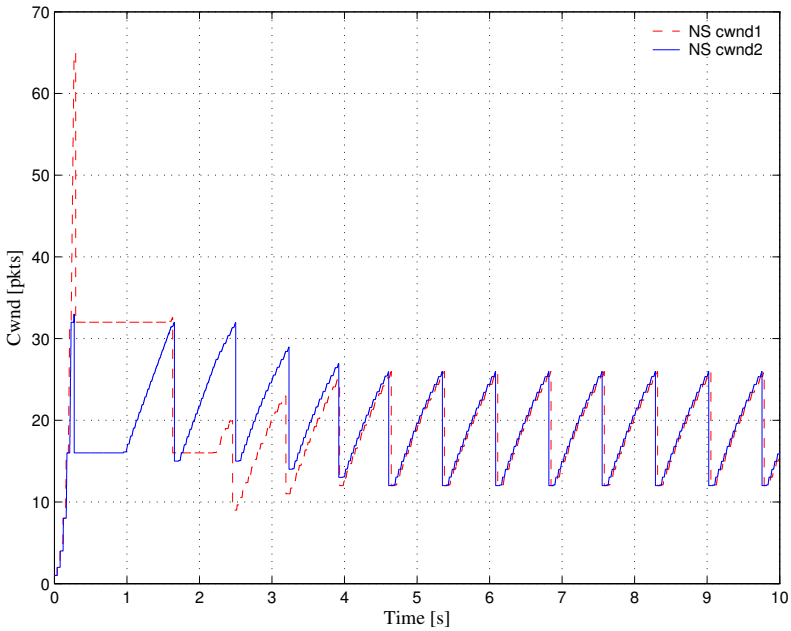

Figure 4.12: Two TCP flows from NS (10Mbit/s, 40ms delay, queue size 17 packets)

which is incremented by one for each train sent. We define a second counter for entrained back-to-back packets m such that

$$m = w_f/2 \tag{4.21}$$

The back-to-back entrainment is modelled at the queue as

$$Q \;=\; \left\{ \begin{array}{ll} (Q \bigoplus f) \bigoplus f & q < qmax - 1 \\ Q \bigoplus f & q < qmax \\ Q & otherwise \end{array} \right\} \tag{4.22}$$

and

$$n_{dropf} \;=\; \left\{ \begin{array}{ll} n_{dropf} & q < qmax - 1 \\ n_{dropf} + 1 & q < qmax \\ n_{dropf} + 2 & otherwise \end{array} \right\} \tag{4.23}$$

We use a line rate timer t_s to space the back-to-back packet trains and send back-to-back packets until $m = 0$. The hybrid automaton for the discrete queue with entrainment can be seen in Figure 4.13.

Figure 4.14 shows a comparison between NS and the hybrid model for one flow. We can see the hybrid model now exhibits similar queue behaviour to the NS model. The extra delays caused by the temporarily filling of the queue causes temporary reduction in the slope of w_f in the hybrid model but w_f still does not match the NS model *cwnd* trace. The entrainment effects in the hybrid model queue are delayed with respect to the NS model and thus the NS model sees a drop before the hybrid model.

It is well known that the Slow Start mode of TCP is difficult to model with a continuous time approximation. We postulate that the discrete nature of the source is best modelled by a discrete model at the TCP source. This type of model could capture the abrupt steps seen during Slow Start and also implement the packet trains at the TCP source rather than attempting to recreate the effects at the queue.

Congestion Avoidance

The entrainment for Congestion Avoidance is implemented by sending trains of packets to the queue as w_f crosses integer boundaries. We pass two back-to-back packets into

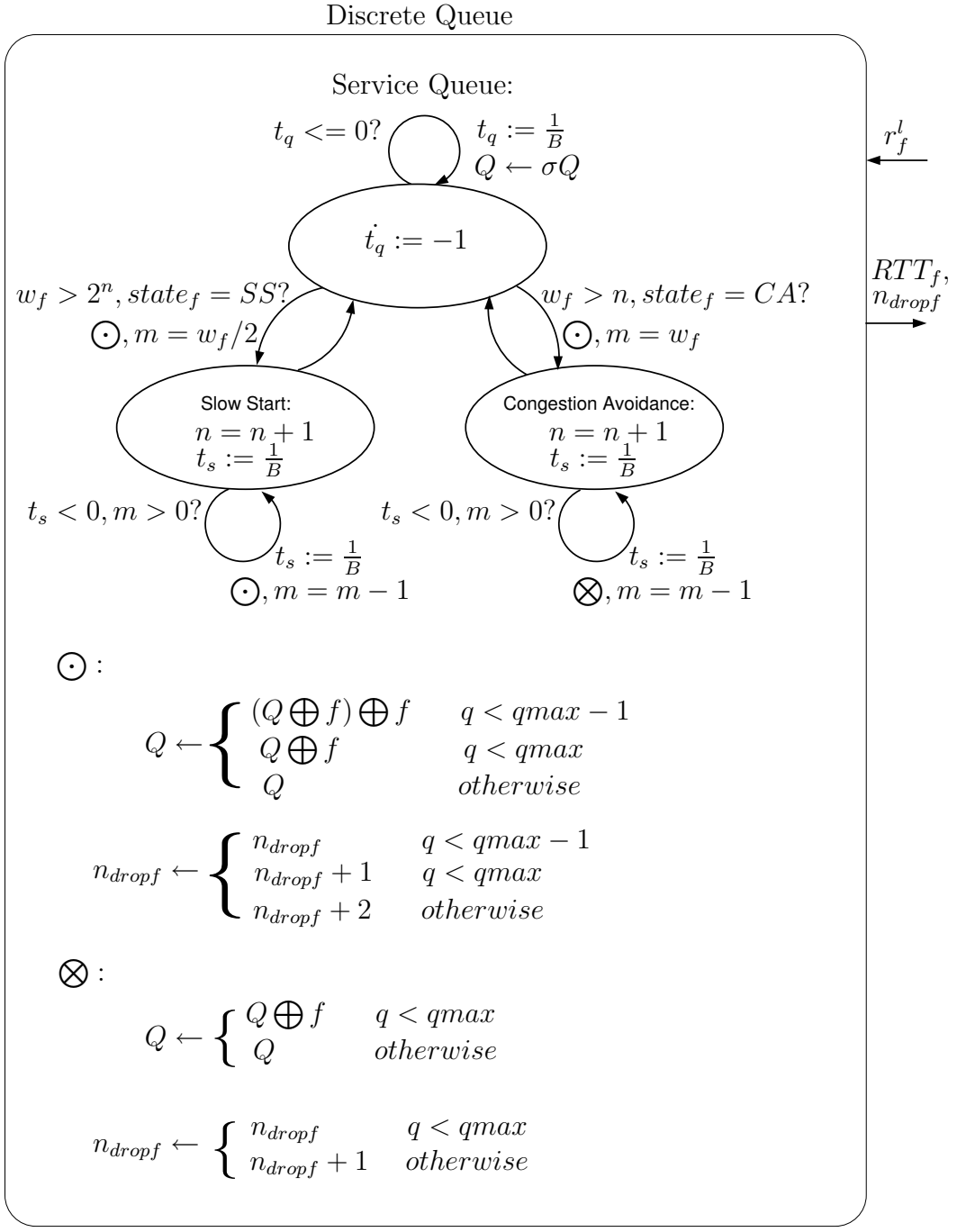

Figure 4.13: Discrete queue with entrainment for Hybrid model (10Mbit/s, 40ms delay, queue size 17 packets)

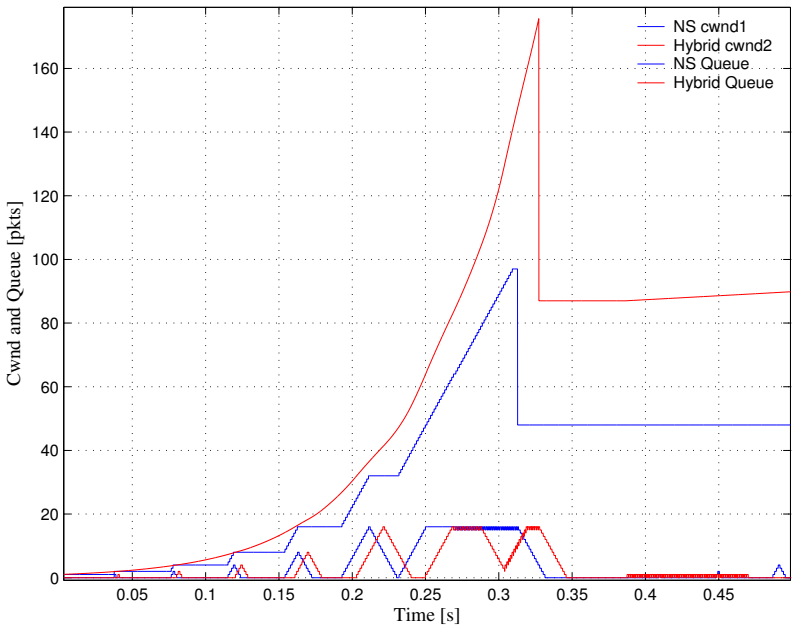

Figure 4.14: Comparison of Slow Start between NS and the hybrid model with discrete queue and full entrainment (10Mbit/s, 40ms delay, queue size 17 packets)

the queue, delay by the line rate and send the remaining packets also delayed by the line rate. Thus one packet is serviced and one packet is queued initially. We send a train of packets to the queue when

$$w_f > n, state_f = CA \tag{4.24}$$

where $state_f$ is the current source mode for flow f and n is a counter initially set to one and incremented by one for each train sent. We define a second counter for entrained packets m such that

$$m = w_f \tag{4.25}$$

The back-to-back entrainment is modelled at the queue as in (4.22) and (4.23)

We use a line rate timer t_s to space the entrained packet and send packets until $m = 0$. The hybrid automaton for the discrete queue with entrainment can be seen in Figure 4.13.

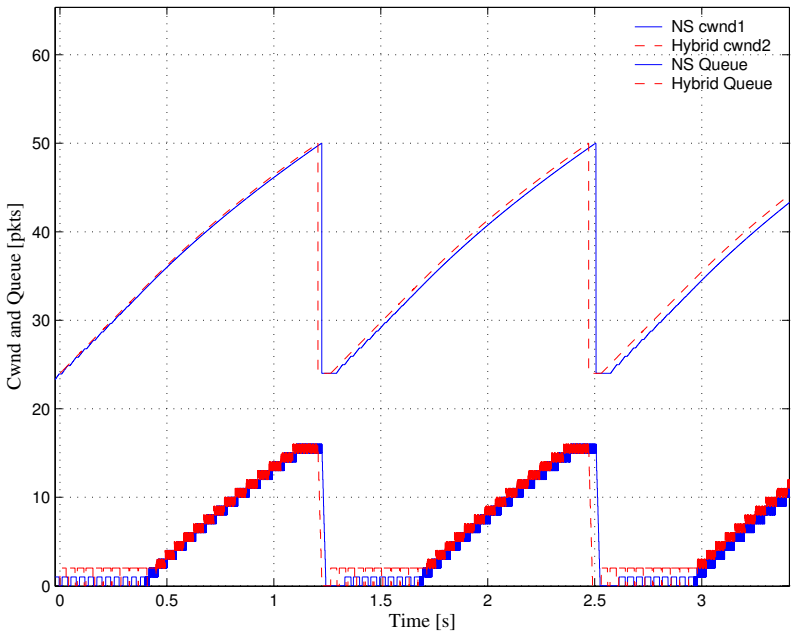

Figure 4.15: Comparison of Congestion Avoidance between NS and the hybrid model with discrete queue and full entrainment (10Mbit/s, 40ms delay, queue size 17 packets)

Figure 4.15 shows a comparison between NS and the hybrid model for one flow. We can see the hybrid model now exhibits similar queue behaviour to the NS model. The comparison for two flows is shown in Figure 4.16. We note that the flows remain synchronised. We also note from Figure 4.17 , which is a closeup of a section of Figure 4.16, that the drops for each flow are separated by exactly half an RTT as would be expected in the entrained case[4].

4.3.3 Discrete Queue with Back-to-back Packets

While entrainment evidently gives rise to synchronised drops, it is unclear whether entrainment is necessary for synchronisation to occur. Rather than model full entrainment of packets, we therefore investigated a simpler model where two back-to-back packets are introduced when *cwnd* worth of packets have successfully been transmitted. We implemented a change to the discrete queue model to emulate this behaviour so that after *cwnd* packets we have

[4]The difference in initial values at the start of each cycle is due to the congestion window of one flow crossing through an integer boundary at 26.

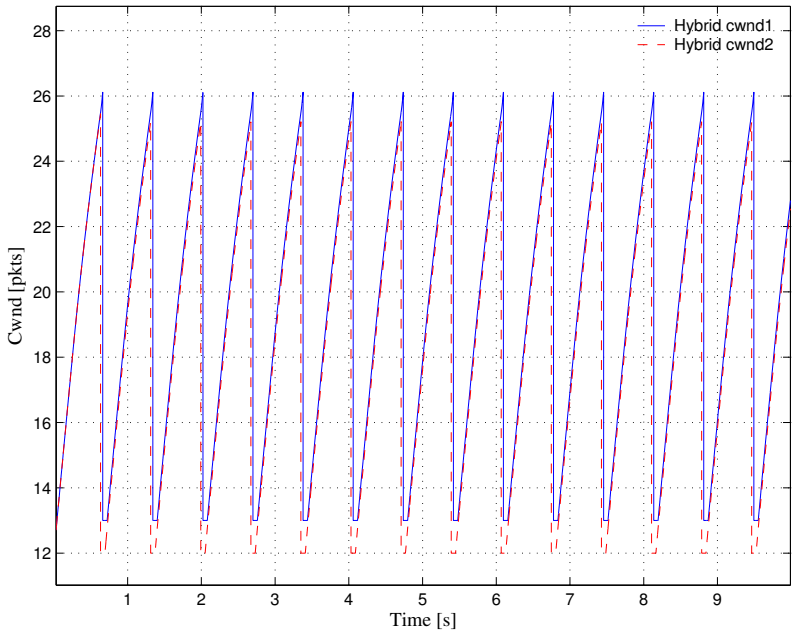

Figure 4.16: Two TCP flows in Congestion Avoidance from the hybrid discrete queue model and full entrainment (10Mbit/s, 40ms delay, queue size 17 packets)

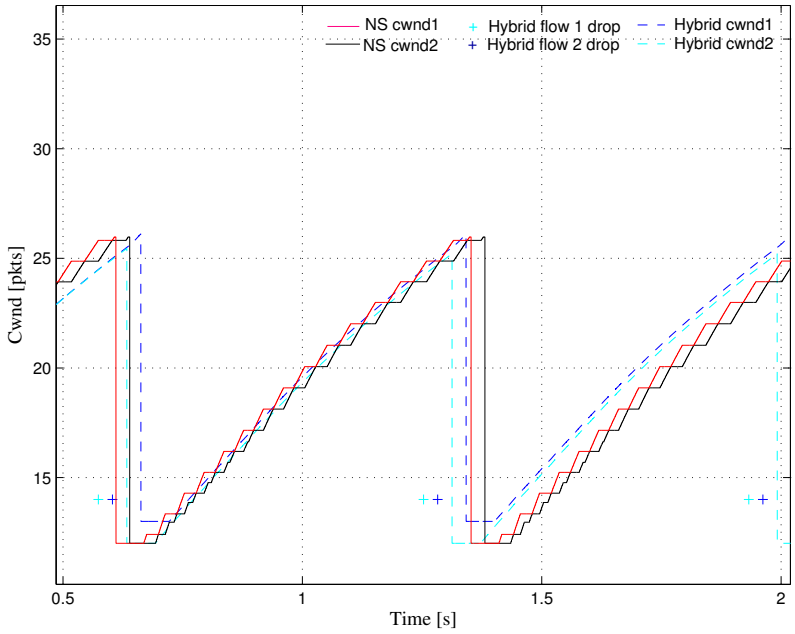

Figure 4.17: Close up comparison of two TCP flows in Congestion Avoidance in NS with the hybrid discrete queue model and full entrainment (10Mbit/s, 40ms delay, queue size 17 packets)

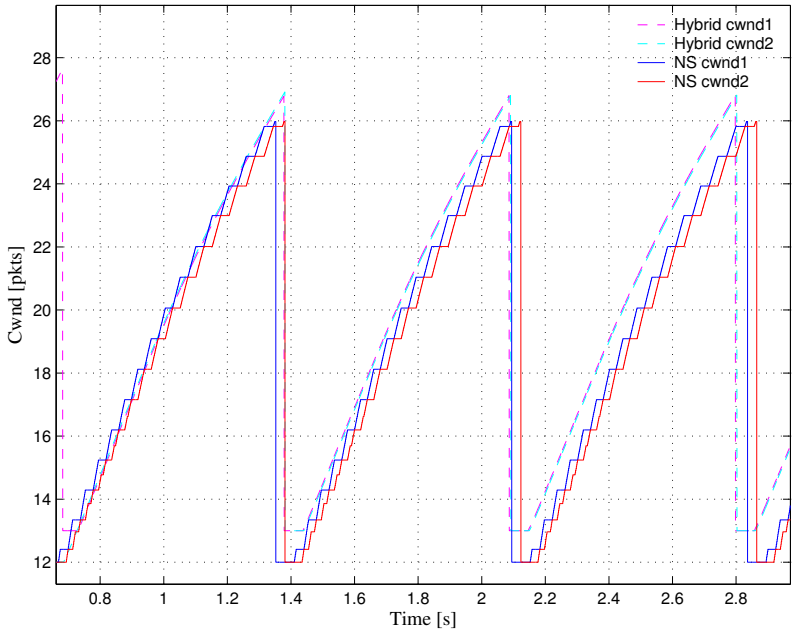

Figure 4.18: Two TCP flows from the hybrid discrete queue model and back-to-back entrainment (10Mbit/s, 40ms delay, queue size 17 packets)

$$(Q \bigoplus f) \bigoplus f, \quad p_f^l = p_f^l - 2 \tag{4.26}$$

Matlab code for discrete queue with entrainment is shown in Appendix A.4.3. As can be seen in Figure 4.18 this simplified model does generate synchronisation of the flows, observe that there is a drift in the congestion windows histories over time relative to NS, similar to that observed with the Bohacek et al model (see Figure 4.8).

4.3.4 Non Synchronised Flows

The forgoing discussion relates to situations where packet drops are synchronised. In this section we briefly assess the accuracy of our models when drops are not synchronised.

Discrete Queue with entrainment:

We consider two flows, the first flow delayed by 60ms and the second flow delayed by 40ms. Figure 4.19 shows the results for NS. Figure 4.20 shows the results for the hybrid model with discrete queue and entrainment. Evidently the congestion window time history is different from the NS congestion window time history of Figure 4.19 and the hybrid model

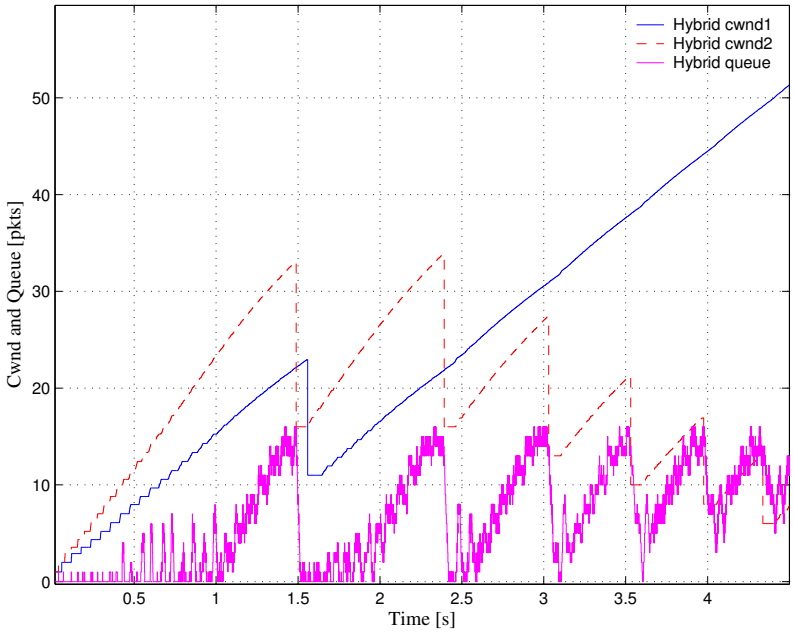

Figure 4.19: Two non synchronised TCP flows in Congestion Avoidance from NS (10Mbit/s, flow 1 60ms delay, flow 2 40ms delay, queue size 17 packets)

does not accurately capture the behaviour in the non-synchronised case.

Discrete Queue with back-to-back packets:

We consider two flows with a variety of different delays in a dumbbell network with a 10Mbit/s link and a queue size of 40 packets. We can see in Figures 4.21, 4.22 and 4.23 that the model does capture the short-term behaviour reasonably well, but that in the longer term it is less accurate.

4.4 Summary

The hybrid fluid model proposed by Bohacek et al [4] was compared to NS. We found that the hybrid model failed to accurately capture the evolution of TCP's congestion control window during Slow Start. This was owing to important packet level effects not included in the hybrid model. The hybrid model was found to predict the congestion window evolution during Congestion Avoidance fairly accurately (small differences in the slope and peak values) under synchronised dropping. The hybrid drop model is not suited to modelling non-synchronised drops for small numbers of flows (Bohacek et al propose a stochastic drop model). The discrete nature of the queue in the network plays a key role

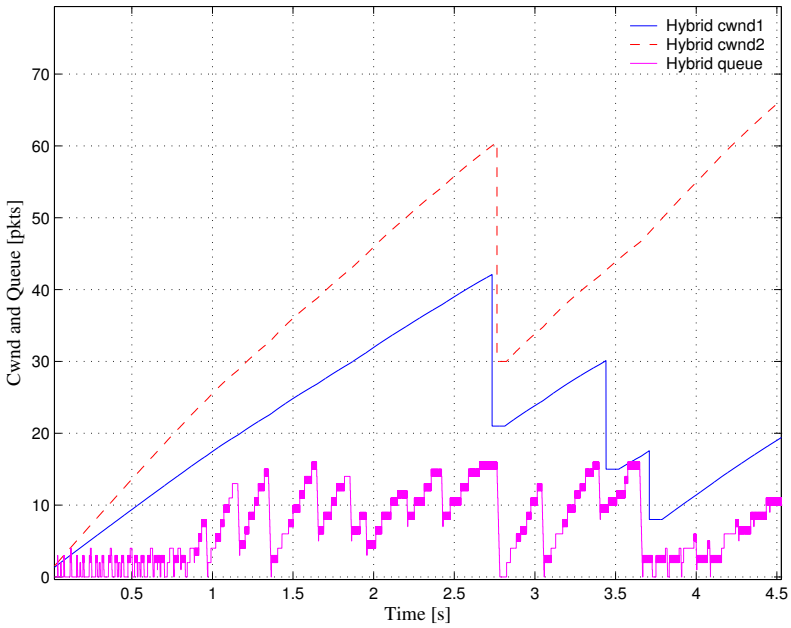

Figure 4.20: Two non synchronised TCP flows in Congestion Avoidance from the hybrid discrete queue model and full entrainment (10Mbit/s, flow 1 60ms delay, flow 2 40ms delay, queue size 17 packets)

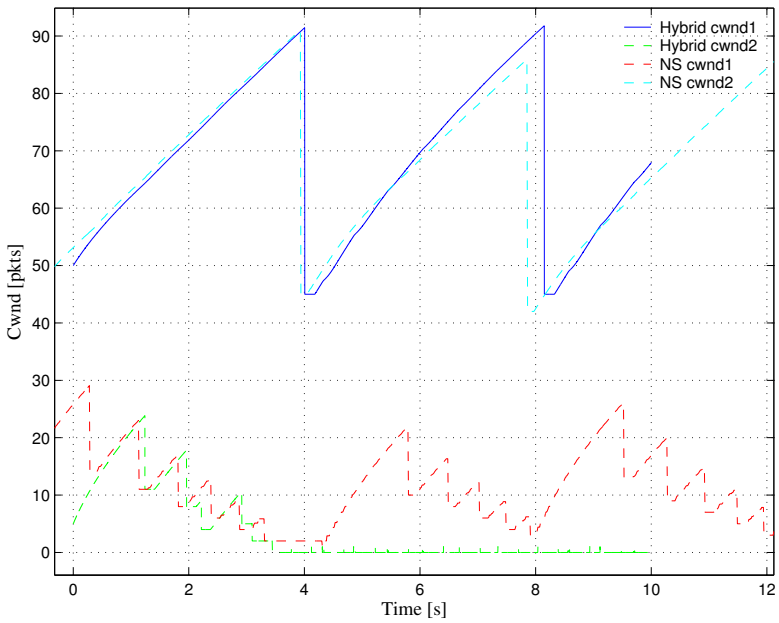

Figure 4.21: Comparison of two non synchronised TCP flows in Congestion Avoidance in NS with the Hybrid Model with Discrete Queue and Back-to-back Packets (10Mbit/s, flow 1 60ms delay, flow 2 40ms delay, queue size 40 packets)

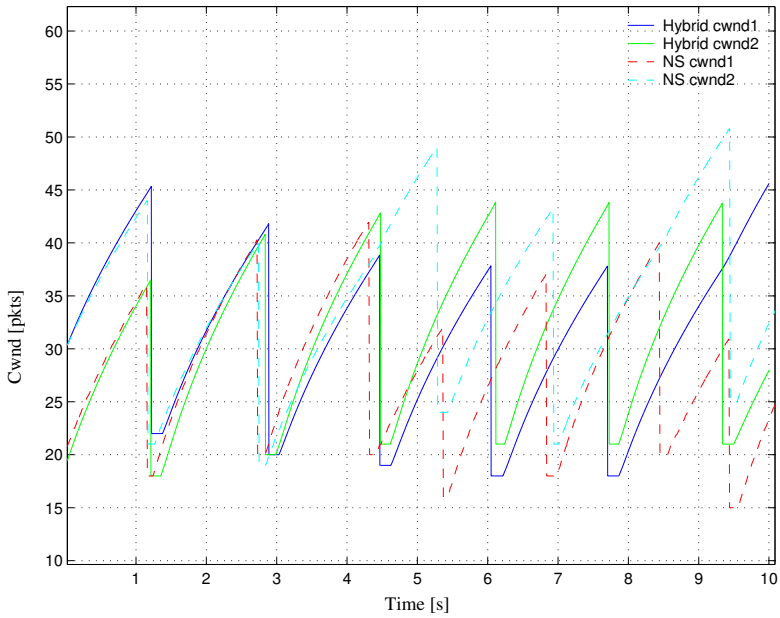

Figure 4.22: Comparison of two non synchronised TCP flows in Congestion Avoidance in NS with the Hybrid Model with Discrete Queue and Back-to-back Packets (10Mbit/s, flow 1 50ms delay, flow 2 40ms delay, queue size 40 packets)

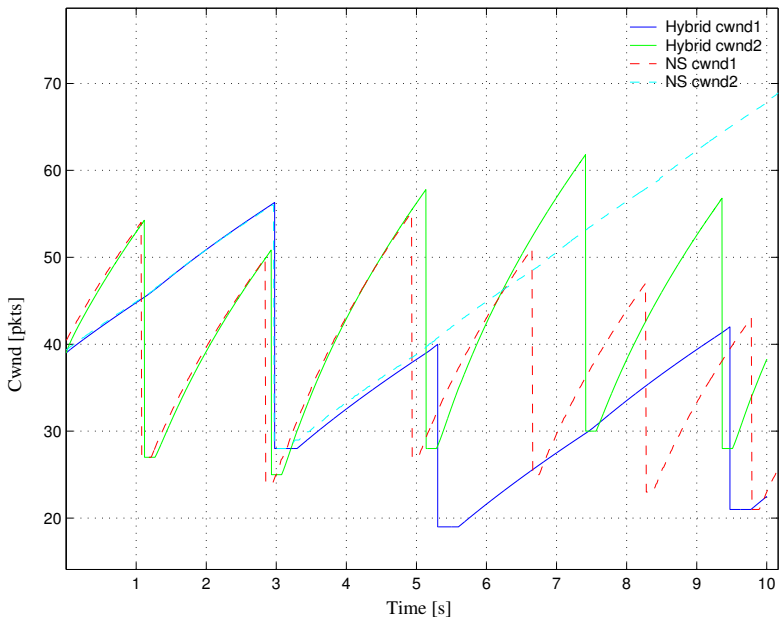

Figure 4.23: Comparison of two non synchronised TCP flows in Congestion Avoidance in NS with the Hybrid Model with Discrete Queue and Back-to-back Packets (10Mbit/s, flow 1 140ms delay, flow 2 40ms delay, queue size 40 packets)

in determining non-synchronised drops. We investigated the use of a discrete queue model in conjunction with the fluid source model proposed by Bohacek et al. It was found that flow synchronisation was not captured by the modified model unless back-to-back packet sequences were included, thereby gaining new insight into the mechanism underlying drop synchronisation. However, we found that this model was insufficient to accurately capture the long term evolution of TCP's congestion control window in non synchronised dropping regimes.

Chapter 5

Analysis and Design of Synchronised Communication Networks

We have seen in the last chapter that a hybrid dynamic model can provide a good approximation to the Congestion Avoidance mode of TCP. In this chapter we concentrate on the dumbbell topology and the case of synchronised flows. We model a network using the TCP AIMD congestion control algorithm as a positive linear system. Taking a first principles approach, we show that such such a network possesses a unique equilibrium and that this equilibrium is globally convergent. Using these results we also establish conditions for fair co-existence of traffic in networks employing a mix of AIMD algorithms.

The chapter is organised as follows. In section 5.1 we define some mathematical preliminaries. Section 5.2 generates the network model and provide a mathematical insight into our network model. In section 5.3 we analyse the convergence and fairness of our model. Section 5.4 verifies our findings using NS simulations. Our findings are summarised in section 5.5.

5.1 Definitions and Mathematical Preliminaries

Throughout, the following notation is adopted: $\mathbb{R}$ denotes the real numbers; $\mathbb{R}^n$ denotes the n-dimensional real Euclidean space; $\mathbb{R}^{n\times n}$ denotes the space of $n \times n$ matrices with real entries; x_i denotes the i^{th} component of the vector x in $\mathbb{R}^n$; a_{ij} denotes the entry in the (i, j) position of the matrix A in $\mathbb{R}^{n\times n}$. We use the symbol $\succ$ to denote that the entries of a matrix (vector) are greater than zero. We say that the matrix A is strictly positive if all entries of the matrix are positive; namely, $A \succ 0$.

Strictly positive matrices will play an important role in developing the results. We note

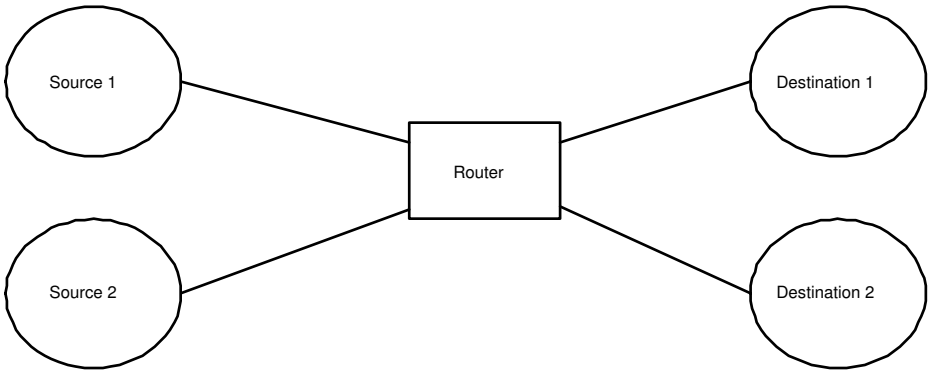

Figure 5.1: Test Network Topology

the following important theorem for strictly positive matrices.

Theorem 5.1.1 *[26, 27, 28] Let $A \in \mathbb{R}^{n\times n}$ be a strictly positive matrix. Then: (i) there is an eigenvalue $\rho(A)$ that is simple and whose magnitude is greater that any other eigenvalue; (ii) A has a positive eigenvector x_p corresponding to $\rho(A)$ and any non-negative eigenvector of A is a multiple of x_p; and (iii)* $\lim_{N\to\infty} \frac{1}{\rho(A)} A^N = x_p y_p^T$ *where* $A^T y_p = \rho(A) y_p$, $x_p \succ 0, y_p \succ 0$, *and* $x_p^T y_p = 1$.

Corollary 5.1.1 *Let $A \in \mathbb{R}^{n\times n}$ with $A \succ 0$. There exists a unique vector x_p such that $Ax_p = \rho(A)x_p, x_p \succ 0$ and $\sum_{i=1}^{n} x_p = 1$. We refer to x_p as the Perron eigenvector of the matrix A and $\rho(A)$ as the Perron eigenvalue of the matrix A.*

A requirement for the design of new protocols is that of fairness. We concentrate here on the following definition of *fairness*.

Definition 5.1.1 *Fairness: Fairness is defined as the requirement that n users, competing for a bandwidth of B packets per second, should be allocated, under ideal conditions, a bandwidth of B/n packets per second per user.*

5.2 A Network Model

We will use the standard dumbbell, see Figure 5.1 model (assuming drop-tail queueing) for our network model and concentrate on the Congestion Avoidance mode for our analysis. We have that

$$cwnd_i \rightarrow cwnd_i + \alpha_i / cwnd_i \tag{5.1}$$

where $\alpha_i = 1$ for standard TCP. On detecting a loss the source enters Fast Recovery mode. The lost packets are retransmitted and the window size $cwnd_i$ of source i is reduced according to

$$cwnd_i \rightarrow \beta_i cwnd_i \tag{5.2}$$

where $\beta_i = 0.5$ for standard TCP . It is assumed that multiple drops within a single round-trip time lead to a single back-off action. When receipt of the retransmitted lost packets is eventually confirmed by the destination, the source re-enters the Congestion Avoidance mode, adjusting its window size according to (5.1). A typical window evolution is depicted in Figure 5.2 ($cwnd_i$ at the time of detecting congestion is denoted by w_i in this figure).

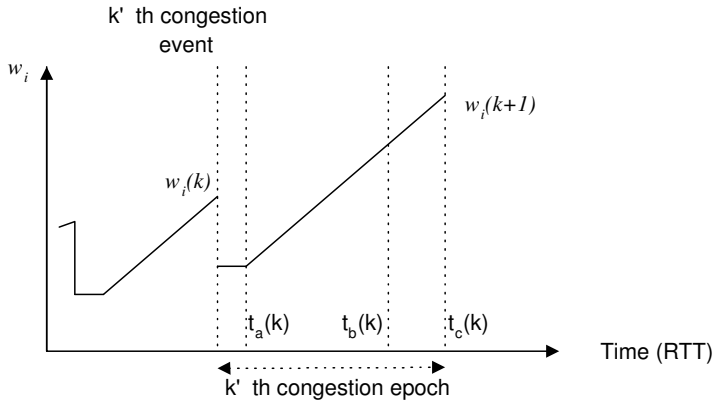

Figure 5.2: Evolution of window size

Over the kth congestion epoch three important events can be discerned: indicated by $t_a(k), t_b(k)$ and $t_c(k)$ in Figure 5.2. The time $t_a(k)$ is the time at which the number of unacknowledged packets in the pipe equals $\beta_i w_i(k)$; $t_b(k)$ is the time at which the pipe is full so that any packets subsequently added will be dropped at the congested queue; $t_c(k)$ is the time at which packet drop is detected by the sources. Note that we measure time in units of round-trip time (RTT).

5.2.1 Model of a Network under Synchronisation

We consider a network of n AIMD sources. We parameterized each source by an additive increase parameter and a multiplicative decrease factor, denoted α_i and β_i respectively.

These parameters satisfy $\alpha_i > 0$ and $0 < \beta_i < 1\ \forall i \in \{1, ..., n\}$. We assume that the event times t_a, t_b and t_c indicated in Figure 5.2 are the same for every source i.e. that the sources are synchronised. This synchronisation condition is valid, for example, when sources are constrained by a shared congested link, the round-trip propagation delay between each source and destination is identical and each source transmits at least one extra packet per round-trip time (i.e. $\alpha_i \geq 1$).

Let $w_i(k)$ denote congestion window size of source i immediately before the kth network congestion event is detected by the sources, see Figure 5.2. It follows from the definition of the AIMD algorithm that the window evolution is completely defined over all time instants by knowledge of the $w_i(k)$ and the event times $t_a(k), t_b(k)$ and $t_c(k)$ of each congestion epoch. We therefore only need to investigate the behaviour of these quantities.

We have that $t_c(k) - t_b(k) = 1$; namely, each source is informed of congestion exactly one RTT after the first dropped packet was transmitted. Also,

$$w_i \geq 0, \sum_{i=1}^{n} w_i = P + \sum_{i=1}^{n} \alpha_i \tag{5.3}$$

where P is the maximum number of packets which can be held in the 'pipe'; this is usually equal to $q_{max} + BT$ where q_{max} is the maximum queue length of the congested link, B is the service rate in packets per second and T is the round-trip time. At the $(k+1)$th congestion event

$$w_i(k+1) \quad = \quad \beta_i w_i(k) + \alpha_i [t_c(k) - t_a(k)]. \tag{5.4}$$

and

$$t_c(k) - t_a(k) \quad = \quad \frac{1}{\sum_{i=1}^{n} \alpha_i} [P - \sum_{i=1}^{n} \beta_i w_i(k)] + 1 \tag{5.5}$$

Substituting into (5.5) from (5.3) yields

$$t_c(k) - t_a(k) \quad = \quad \frac{1}{\sum_{i=1}^{n} \alpha_i} [\sum_{i=1}^{n} (1 - \beta_i) w_i(k)] \tag{5.6}$$

Hence,

$$w_i(k+1) \quad = \quad \beta_i w_i(k) + \frac{\alpha_i}{\sum_{j=1}^{n} \alpha_i} [\sum_{i=1}^{n} (1 - \beta_i) w_i(k)] \tag{5.7}$$

The dynamics of the entire network can be described by writing all n equations in matrix

form:

$$W(k+1) = AW(k) \tag{5.8}$$

where $W^T(k) = [w_1(k), \cdots, w_n(k)]$, and

$$\begin{aligned} A &= \begin{bmatrix} \beta_1 & 0 & \cdots & 0 \\ 0 & \beta_2 & 0 & 0 \\ \vdots & 0 & \ddots & 0 \\ 0 & 0 & \cdots & \beta_n \end{bmatrix} + \frac{1}{\sum_{j=1}^n \alpha_i} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \cdots \\ \alpha_n \end{bmatrix} \begin{bmatrix} 1-\beta_1 & 1-\beta_2 & \cdots & 1-\beta_n \end{bmatrix} \\ &= diag[\beta_i] + \frac{1}{\sum_{j=1}^n \alpha_i} g^T h, \end{aligned} \tag{5.9}$$

with $g^T = [\alpha_1, \alpha_2, \cdots, \alpha_n]$ and $h^T = [1-\beta_1, 1-\beta_2, \cdots, 1-\beta_n]$. Note that the initial condition $W(0)$ is subject to constraint (5.3) (this simply ensures that the window sizes specified by $W(0)$ are non-negative and correspond to a congestion event).

It follows that the synchronised network (5.8) is a positive linear system and that the matrix A is strictly positive: $A \succ 0$ since $0 < \beta_i < 1$, $\forall\, i \in \{1, ..., n\}$.

Comment: We note that this model incorporates a number of key features of real networks: the hybrid nature of AIMD algorithms; time-varying communication delays on links; and drop-tail queueing.

Comment (Convergence): We note the two following cases where convergence, measured in number of congestion epochs, does not depend on the network α_i [5].

(i) All of the sources share the same increase parameter: $\alpha_1 = \alpha_2 = \cdots \alpha_n = \alpha$, and

$$A = diag[\beta_i] + \frac{1}{n} \begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix} h \tag{5.10}$$

Under these conditions the network dynamics (and so the rate of convergence) depends solely on the decrease parameters β_i.

(ii) All of the sources share the same decrease parameter (the α_i need not be the same for all sources): $\beta_1 = \beta_2 = \cdots = \beta_n = \beta$, and the eigenvalues of A (other than the Perron eigenvalue) have value β. Thus, the rate of convergence to a fixed point is β^k where k is the congestion epoch and using this it follows, for example, that the 95% rise time is

$$\log 0.05 / \log \beta \tag{5.11}$$

giving a rise time of 4 congestion epochs when $\beta = 0.5$. Note that the ratio of the α_i to β determines the duration of the congestion epochs according to the relation

(5.6).

5.3 Convergence and Fairness

We now present the main mathematical results.

Theorem 5.3.1 *Let A be defined as in Equation (5.9). Then: (i) the Perron eigenvalue of A is given by $\rho(A) = 1$; (ii) the Perron eigenvector of A is given by $x_p^T = \gamma[\frac{\alpha_1}{1-\beta_1}, ..., \frac{\alpha_n}{1-\beta_n}]$, where $\sum_{i=1}^{n} \gamma x_{pi} = 1$.*

Proof : Since A is a positive matrix, the Perron eigenvector is the only positive eigenvector. It follows by inspection that x_p is the Perron eigenvector and $\rho(A) = 1$.

Corollary 5.3.1 *For a network of synchronised time-invariant AIMD sources: (i) the network has a Perron eigenvector $x_p^T = \gamma[\frac{\alpha_1}{1-\beta_1}, ..., \frac{\alpha_n}{1-\beta_n}]$; and (ii) the Perron eigenvalue is $\rho(A) = 1$. It follows from Theorem 5.1.1 that all other eigenvalues of A satisfy $|\lambda_i(A)| < \rho(A)$. The network possesses a unique stationary point $W_{ss} = \Theta x_p$, where Θ is a positive constant. The stationary point is globally attractive, i.e. $\lim_{k\to\infty} W(k) = \Theta x_p$, and the rate of convergence of the network to W_{ss} depends upon the second largest eigenvalue of A.*

Proof : We are interested in the evolution of the system

$$W(k+1) = AW(k), \tag{5.12}$$

where A is defined by (5.9) and the initial condition $W(0)$ is subject to constraint (5.3) (this ensures that W(0) corresponds to a congestion event). The convergence of this system is determined by

$$\lim_{k\to\infty} W(k) = \lim_{k\to\infty} A^k W(0),$$

From Theorem (5.1.1) we have

$$\lim_{k\to\infty} \frac{1}{\rho(A)} A^k = x_p y_p^T,$$

where x_p is the Perron eigenvector of A, and y_p is the associated eigenvector of A^T. By inspection, $y_p^T = [1, 1, ..., 1]$. Hence, it follows that

$$\lim_{k\to\infty} W(k) \quad = \quad \lim_{k\to\infty} A^k W(0)$$

$$
\begin{aligned}
&= x_p y_p^T W(0), \\
&= \Theta x_p,\ \Theta = y_p^T W(0) = \sum_{i=1}^{n} w_i(0).
\end{aligned}
$$

Owing to constraint (5.3), Θ is a constant, independent of $W(0)$. The rate of convergence of A^k depends on the second largest eigenvalue of A (see item (j) on page 498 in [27]).

Comment(Fair allocation of network bandwidth): Let $\alpha_i = \lambda(1 - \beta_i)\ \forall i$ and for some $\lambda > 0$. Then $W_{ss}^T = \Theta/n\ [1, 1, ..., 1]$ i.e. $w_1 = w_2 = ... = w_n$. For networks where the queueing delay is small relative to the propagation delay, the send rate is essentially proportional to the window size. In this case, it can be seen that $\alpha_i = \lambda(1-\beta)\forall i \in \{1, .., n\}$ is a condition for a fair allocation of network bandwidth. For the standard TCP choices of $\alpha = 1$ and $\beta = 0.5$, we have $\lambda = 2$ and the condition for other AIMD flows to co-exist fairly with TCP is that they satisfy

$$\alpha_i = 2(1 - \beta_i). \tag{5.13}$$

Comment: This equation provides a very useful basis for allocation of bandwidth between flows. It allows for a network of TCP sources with different α's and β's where fairness can be guaranteed.

5.4 Verification of Predictions through Simulation

The mathematical results above provide some useful insights into the operation of TCP congestion control within a generalised network environment particularly with regard to fairness and convergence. In this section we seek to verify these analytic predictions using NS simulations. We focus on a dumbbell topology with a 10Mbit/s bottleneck link with 20ms delay and a queue size of 17 packets.

5.4.1 Fairness

(i) The analysis predicts fairness of AIMD flows with TCP when every flow on the network select α and β according to (5.13). Initially we consider two standard TCP flows both with $\alpha = 1$ and $\beta = 0.5$. As we can see from Figure 5.3 there is a fair allocation of pipe (same peak window size) between the flows.

We now look at two flows, one a standard TCP flow and the second with values for α and β that obey (5.13) namely $\alpha = 1$ and $\beta = 0.5$. The second flow uses $\alpha = 1.5$ and $\beta = 0.25$. The results for this can be seen in Figure 5.4. Again it can be seen that the pipe is shared fairly between the flows.

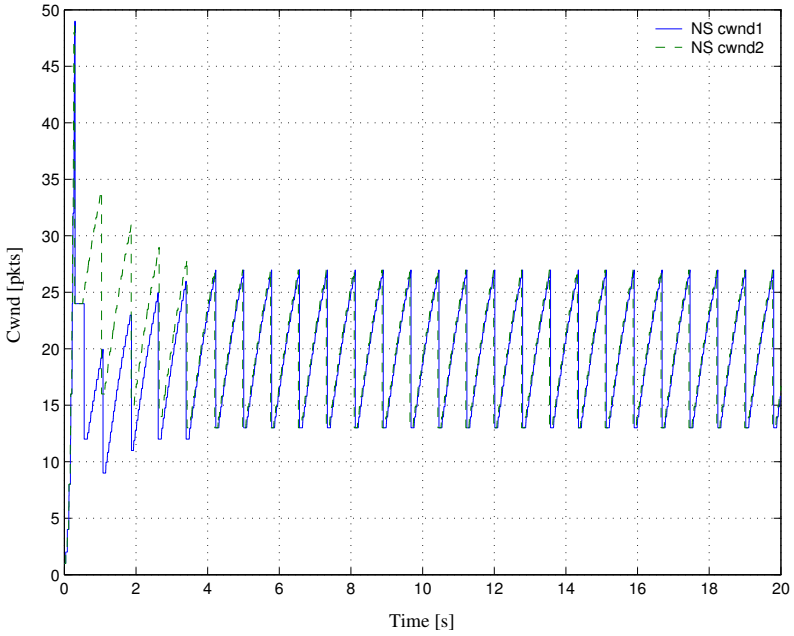

Figure 5.3: Two NS TCP flows ($\alpha_1 = 1, \beta_1 = 0.5, \alpha_2 = 1, \beta_2 = 0.5$) (10Mbit/s, 40ms delay, queue size 17 packets)

(ii) More generally the analysis predicts that the peak window sizes of the two flows will be in the ratio $\alpha_1(1-\beta_2)/\alpha_2(1-\beta_1)$. Consider therefore the case where α and β do not obey (5.13). Flow one is standard TCP with $\alpha = 1$ and $\beta = 0.5$ and flow two uses $\alpha = 2$ and $\beta = 0.5$. We can see from Figure 5.5 that varying α and not adhering to (5.13) causes an unfair allocation of bandwidth between the flows. The ratio of the peak window size in Figure 5.5 is 0.47, which is in good agreement with the ratio of 0.5 predicted by the theoretical analysis. Additionally if β does not adhere to (5.13), there will also be an unfair allocation of bandwidth between flows. This can be seen in Figure 5.6 where flow one is standard TCP $\alpha = 1$ and $\beta = 0.5$ and flow two uses $\alpha = 1$ and $\beta = 0.75$. The ratio of the peak window size in Figure 5.6 is 0.5, which is identical to the ratio predicted by the theoretical analysis.

5.4.2 Convergence

The analysis indicates that the convergence time (95% rise time measured in congestion epochs) is $log0.05/log\overline{\beta}$ where $\overline{\beta}$ is the second largest eigenvalue of the matrix A. Observe that the congestion time (measured in epochs) is predicted to be independent of the increase parameter α when all flows use the same back-off parameter.

(i) $\beta = 0.5$. The analysis predicts that convergence should take just over four congestion

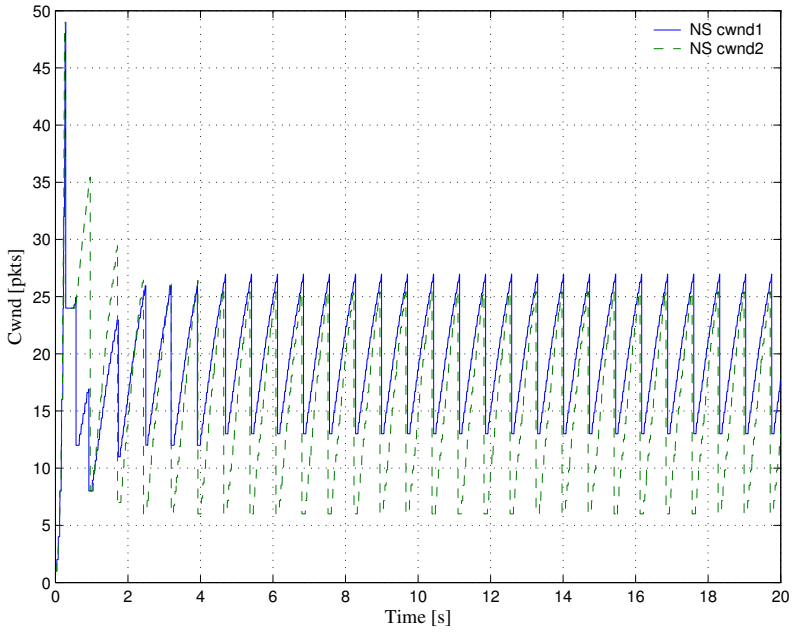

Figure 5.4: Two NS TCP flows ($\alpha_1 = 1, \beta_1 = 0.5, \alpha_2 = 1.5, \beta_2 = 0.25$) (10Mbit/s, 40ms delay, queue size 17 packets)

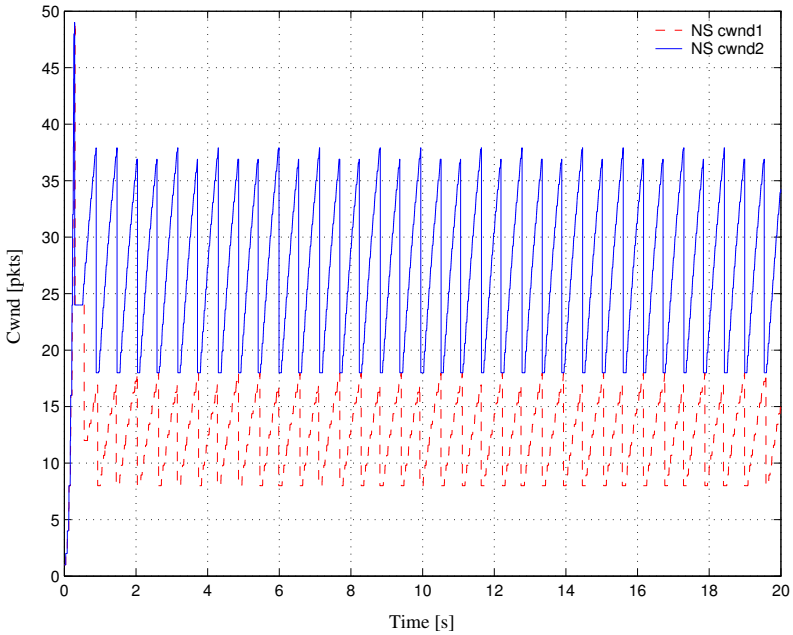

Figure 5.5: Two NS TCP flows ($\alpha_1 = 1, \beta_1 = 0.5, \alpha_2 = 2, \beta_2 = 0.5$) (10Mbit/s, 40ms delay, queue size 17 packets)

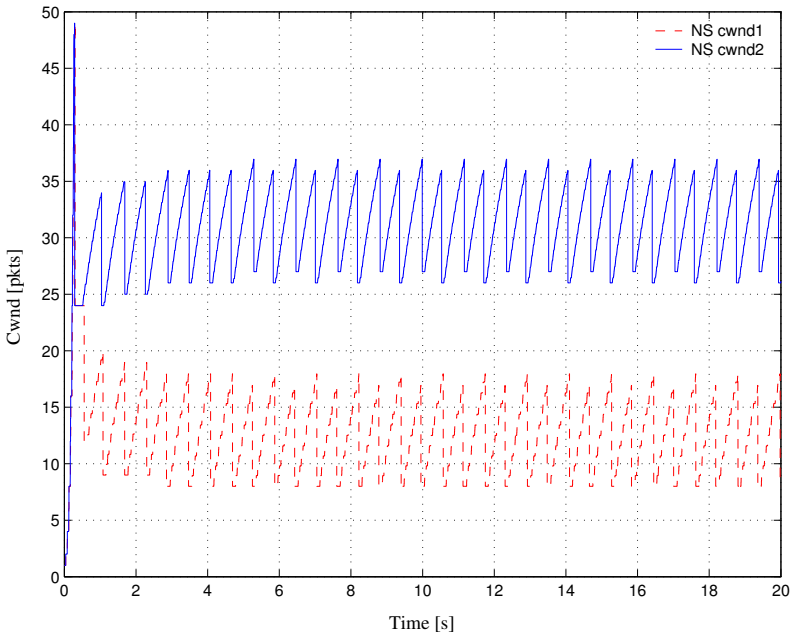

Figure 5.6: Two NS TCP flows ($\alpha_1 = 1, \beta_1 = 0.5, \alpha_2 = 1, \beta_2 = 0.75$) (10Mbit/s, 40ms delay, queue size 17 packets)

cycles when $\beta = 0.5$. Figure 5.7 shows the results for an experiment where the second flow is started five seconds after the first flow. Slow start is turned off for the second flow. We can see that the flows converge after approximately four congestion cycles.

(ii) $\beta = 0.75$. Figure 5.8 shows the same experiment but with $\beta = 0.75$. We can see that the convergence is significantly slower, approximately thirteen congestion cycles. In Figure 5.9 α is changed so that $\alpha = 2$. We can see that it still takes thirteen congestion cycles for the flows to converge and that the value of α does not affect the rate of convergence in terms of congestion epochs (the duration of the epochs does of course decrease as α becomes larger).

(iii) Mixed β. We now consider the case where we have a mixture of flows with different backoff ratios. From [27] we can see that the rate of convergence is dictated by the second largest eigenvalue of A. By using two flows with the same, high, values of β, we can affect the rate of convergence for all flows. This occurs as we are forcing the second and third eigenvalue of A to be at this high β value. We construct an experiment with five flows, three flows starting at time zero and two after five seconds. For clarity, we only show one flow from each grouping as the others are the same. Initially we have five standard TCP flows with $\alpha = 1$ and $\beta = 0.5$. We can see from Figure 5.10 that the rate of convergence is around four epochs as expected. We now change the β for the two delayed flows. The first three flows are standard TCP flows with $\alpha = 1$ and $\beta = 0.5$. The second two flows

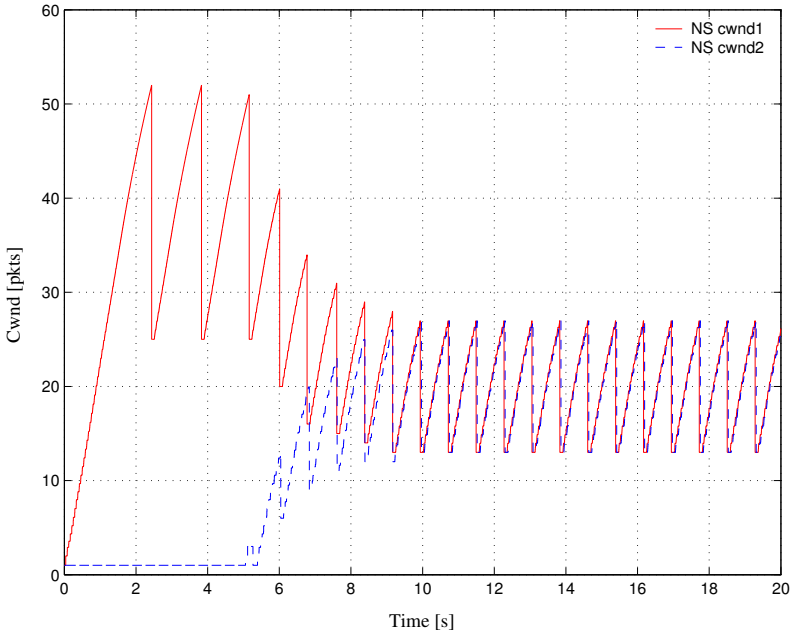

Figure 5.7: Two NS TCP flows ($\alpha_1 = 1, \beta_1 = 0.5, \alpha_2 = 1, \beta_2 = 0.5$) , flow 1 starting after 5 seconds (10Mbit/s, 40ms delay, queue size 17 packets)

have $\alpha = 1$ and $\beta = 0.95$. Figure 5.11 show the results of this experiment and we can see that the number of congestion cycles for convergence after the first three flows start is now approximately ten epochs. It is therefore possible with just two flows with high β's to guarantee the rate of convergence in a network with a large number of flows.

5.5 Summary

In this chapter we verified he predictions of a simple mathematical model for AIMD flows under drop synchronisation. We have established the stability of synchronised flows using the model and showed that there is an asymptotic distribution of the network pipe between flows (the Perron eigenvector). We showed that the rate of convergence is dictated by the second largest eigenvalue of the matrix describing the network dynamics. We verified these predictions using NS simulations.

The mathematical framework described in this chapter provides some new insights into the convergence and fairness behaviour of TCP with synchronised flows. The simple formula $\alpha_i = 2(1-\beta_i)$ can be used as a basis of fairness between TCP flows. Additionally we establish that convergence (in terms of congestion epochs) is only dependant on the backoff factor β used in the network and demonstrated that by using large β's in two or

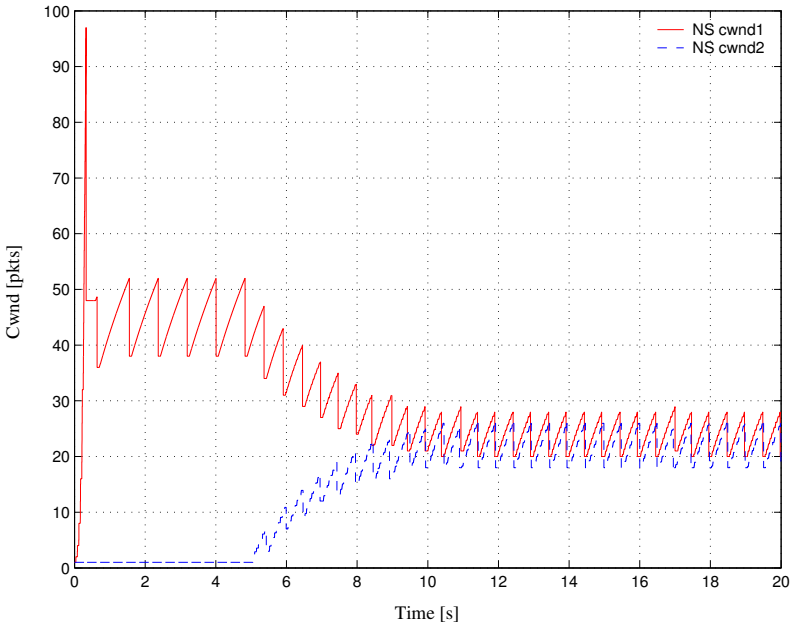

Figure 5.8: Two NS TCP flows ($\alpha_1 = 1, \beta_1 = 0.75, \alpha_2 = 1, \beta_2 = 0.75$) , flow 1 starting after 5 seconds (10Mbit/s, 40ms delay, queue size 17 packets)

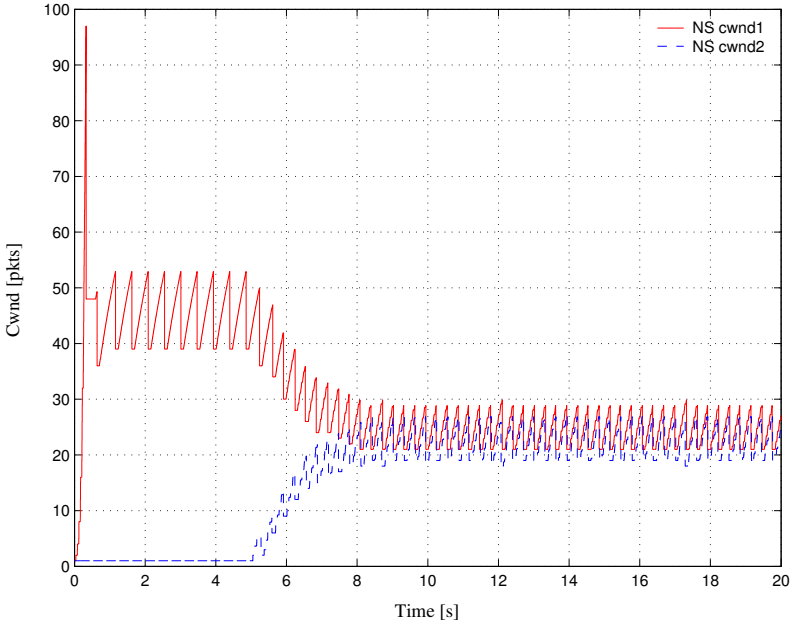

Figure 5.9: Two NS TCP flows ($\alpha_1 = 2, \beta_1 = 0.75, \alpha_2 = 2, \beta_2 = 0.75$) , flow 1 starting after 5 seconds (10Mbit/s, 40ms delay, queue size 17 packets)

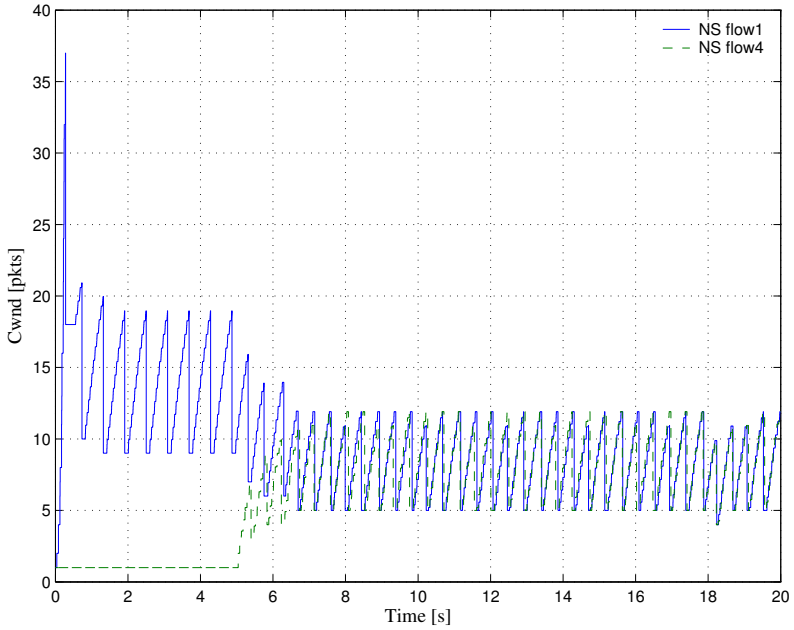

Figure 5.10: Five NS TCP flows, flows 1 and 5 shown (flow 1 - 5 $\alpha = 1, \beta = 0.5$) , flow 4 and 5 starting after 5 seconds (10Mbit/s, 40ms delay, queue size 17 packets)

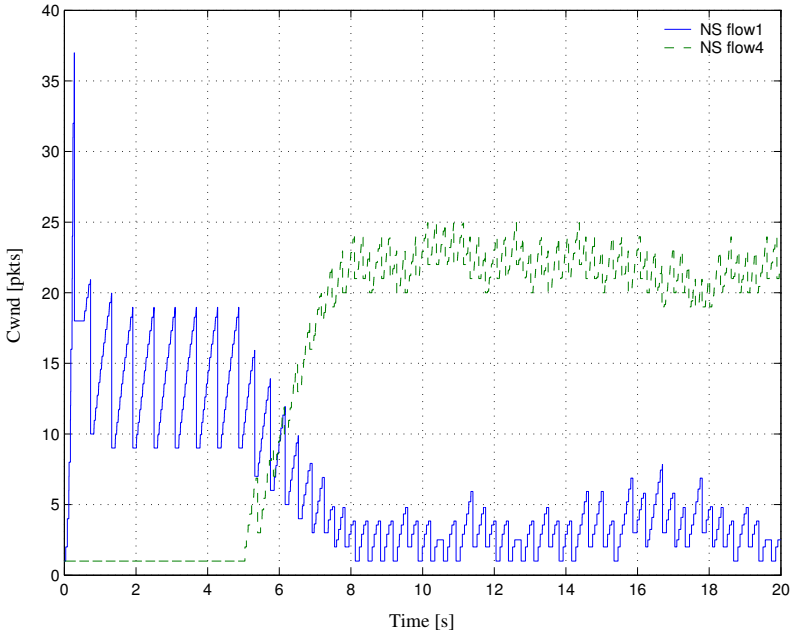

Figure 5.11: Five NS TCP flows, flows 1 and 5 shown (flow 1 - 3 $\alpha = 1, \beta = 0.5$, flow 4 - 5 $\alpha = 1, \beta = 0.95$) , flow 4 and 5 starting after 5 seconds (10Mbit/s, 40ms delay, queue size 17 packets)

more flows we can control the rate of convergence for the whole network.

Chapter 6

Summary and Conclusions

The objective of this thesis is to investigate the development of mathematical models suitable for the analysis and design of TCP congestion control. While there are a multitude of protocols in use in the Internet, the most prevalent is the Transmission Control Protocol (TCP). We review current models for TCP congestion control. After constructing and instrumenting a testbed, we compare the NS packet level model (the de-facto standard for simulation studies) against real TCP implementations. A recently proposed simplified hybrid model is assessed against NS and a number of modifications studied. Based on the insights gained, a simplified mathematical model of TCP congestion avoidance dynamics in synchronised networks is evaluated. Specifically:

- We detailed the TCP source and network conditions used as a basis for our modelling framework. We focused on the dynamics of long lived TCP flows in a dumbbell network topology with a drop-tail queue. We specified mode changes, quantisation effects, entrainment and time varying delays as the features we required for modelling for design.
- A test network was developed and the FreeBSD TCP stack instrumented.
- Using the developed testbed network we assessed the NS simulation model in a variety of network scenarios. It was found that NS provided accurate results for flows with synchronised drops. For the non-synchronised flows and flows with delayed ACK's, NS was found to be less accurate. We demonstrated the presence of entrainment in a our testbed network and on a live network. We confirmed the presence of entrainment and phase effects in NS and our test network.
- The hybrid fluid model proposed by Bohacek et al [4] was compared to NS. We found that the hybrid model failed to accurately capture the evolution of TCP's congestion control window during Slow Start. This was owing to important packet level effects

not included in the hybrid model. The hybrid model was found to predict the congestion window evolution during Congestion Avoidance fairly accurately (small differences in the slope and peak values) under synchronised dropping. The hybrid drop model is not suited to modelling non-synchronised drops for small numbers of flows (Bohacek et al propose a stochastic drop model). The discrete nature of the queue in the network plays a key role in determining non-synchronised drops. We investigated the use of a discrete queue model in conjunction with the fluid source model proposed by Bohacek et al. It was found that flow synchronisation was not captured by the modified model unless back-to-back packet sequences were included, thereby gaining new insight into the mechanism underlying drop synchronisation. However, we found that this model was insufficient to accurately capture the long term evolution of TCP's congestion control window in non synchronised dropping regimes.

- Using the insight gained from the study of the hybrid fluid models, a simple mathematical model for AIMD flows under drop synchronisation is proposed and evaluated. We established the stability of flows using this model and showed that there is an asymptotic distribution of the network pipe between flows. We showed that the rate of convergence is dictated by the second largest eigenvalue of a certain matrix describing the network dynamics. We verified these predictions using NS simulations.

The work performed in this thesis has been a preliminary study of modelling for the design of TCP congestion control networks. Further work is clearly necessary.

Appendix A

Source code

A.1 FreeBSD TCP Kernel Changes

A.1.1 Logging Kernel Data Structures - tcp_logit.h

```
#define TCPLLEN 524288
#define TCPLMASK (TCPLLEN-1)

#define TCPL_RTT 1
#define TCPL_SRTT 2
#define TCPL_CWND 3
#define TCPL_SSTHRESH 4
#define TCPL_WND 5
#define TCPL_STARTTIME 6

extern struct tcplog {
        int type;
        struct timeval tv;
        u_short sport;
        u_short dport;
        union {
                int snd_rtt;
                int snd_srtt;
                u_long snd_cwnd;
                u_long snd_ssthresh;
                u_long snd_wnd;
                tcp_seq starttime;
```

```
        } data;
} tcplog[];

extern unsigned int tcpcurlog;

#define TCP_STASH_RTT(v,w,x) \
        do { \
                struct tcplog *l = &tcplog[tcpcurlog++ & TCPLMASK]; \
                microtime(&(l->tv)); \
                l->type = TCPL_RTT; \
                l->sport = w; \
                l->dport = x; \
                l->data.snd_rtt = v; \
        } while(0);

#define TCP_STASH_SRTT(v,w,x) \
        do { \
                struct tcplog *l = &tcplog[tcpcurlog++ & TCPLMASK]; \
                microtime(&(l->tv)); \
                l->type = TCPL_SRTT; \
                l->sport = w; \
                l->dport = x; \
                l->data.snd_srtt = v; \
        } while(0);

#define TCP_STASH_CWND(v,w,x) \
        do { \
                struct tcplog *l = &tcplog[tcpcurlog++ & TCPLMASK]; \
                microtime(&(l->tv)); \
                l->type = TCPL_CWND; \
                l->sport = w; \
                l->dport = x; \
                l->data.snd_cwnd = v; \
        } while(0);

#define TCP_STASH_SSTHRESH(v,w,x) \
        do { \
                struct tcplog *l = &tcplog[tcpcurlog++ & TCPLMASK]; \
                microtime(&(l->tv)); \
                l->type = TCPL_SSTHRESH; \
                l->sport = w; \
                l->dport = x; \
                l->data.snd_ssthresh = v; \
```

```
        } while(0);

#define TCP_STASH_WND(v,w,x) \
        do { \
                struct tcplog *l = &tcplog[tcpcurlog++ & TCPLMASK]; \
                microtime(&(l->tv)); \
                l->type = TCPL_WND; \
                l->sport = w; \
                l->dport = x; \
                l->data.snd_wnd = v; \
        } while(0);
#define TCP_STASH_STARTTIME(v,w,x) \
        do { \
                struct tcplog *l = &tcplog[tcpcurlog++ & TCPLMASK]; \
                microtime(&(l->tv)); \
                l->type = TCPL_STARTTIME; \
                l->sport = w; \
                l->dport = x; \
                l->data.starttime = v; \
        } while(0);
```

A.1.2 Logging Kernel Sysctl - tcp_logit.c

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/linker_set.h>
#include <sys/sysctl.h>

#include <netinet/in.h>
#include <netinet/in_pcb.h>
#include <netinet/tcp.h>
#include <netinet/tcp_var.h>
#include <netinet/tcp_logit.h>

struct tcplog tcplog[TCPLLEN];
unsigned int tcpcurlog = 0;

SYSCTL_OID(_net_inet_tcp, TCPCTL_LOGIT, tcplog, CTLTYPE_OPAQUE|CTLFLAG_RD,
```

```
        tcplog, sizeof(struct tcplog) * TCPLLEN, sysctl_handle_opaque,
        "S,tcplog", "Our tcp log");
```

A.1.3 Memory Dump of Logged Data- tcplog.c

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/sysctl.h>
typedef u_int32_t tcp_seq;
#include "/usr/src/sys/netinet/tcp_logit.h"
#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char **argv) {
        struct tcplog tl[TCPLLEN];
        size_t len,size;
        int num, i ,mib[2], count=0, port=0, mss=1448;
        struct timeval boottime;
        struct timeval inittime;
        tcp_seq initseq;
        pid_t pid;
        FILE *tempfile;

        if (argc > 1) port = atoi(argv[1]);
        len = sizeof(tl);
        mib[0] = CTL_KERN;
        mib[1] = KERN_BOOTTIME;
        size = sizeof(boottime);
        if (sysctl(mib, 2, &boottime, &size, NULL, 0) == -1) {
                perror("sysctlbyname failed");
                exit(1);
        }

        if (sysctlbyname("net.inet.tcp.tcplog", tl, &len, NULL, NULL) != 0) {
                perror("sysctlbyname failed");
                exit(1);
        }

        num = len/sizeof(tl[0]);
        tempfile = fopen("/tmp/xxx","w");
```

```
        for (i = 0; i < num; i++) {
        if (port == 0 || (tl[i].sport == atoi(argv[1]) || tl[i].dport == atoi(ar
gv[1]))) {
                fprintf(tempfile,"%f %d %d %d ",
                        tl[i].tv.tv_sec +
                        1E-6 * tl[i].tv.tv_usec,
                        tl[i].type,tl[i].sport,tl[i].dport);
                switch (tl[i].type) {
                case TCPL_RTT:
                        fprintf(tempfile,"%d RTT", tl[i].data.snd_rtt);
                        /* in ticks */
                        break;
                case TCPL_SRTT:
                        fprintf(tempfile,"%d RTT", tl[i].data.snd_srtt);
                        /* in ticks */
                        break;
                case TCPL_CWND:
                        fprintf(tempfile,"%f CWND", (float) tl[i].data.snd_cwnd/
mss);
                        /* in packets */
                        break;
                case TCPL_SSTHRESH:
                        fprintf(tempfile,"%f SSTHRESH", (float) tl[i].data.snd_s
sthresh/mss);
                        /* in packets */
                        break;
                case TCPL_WND:
                        fprintf(tempfile,"%f WND", (float) tl[i].data.snd_cwnd/m
ss);
                        /* in packets */
                        break;
                case TCPL_STARTTIME:
                        if (tl[i].sport == 9)
                        fprintf(tempfile,"%f ACK", (float) tl[i].data.starttime/
mss);
                        else
                        fprintf(tempfile,"%f SND", (float) tl[i].data.starttime/
mss);
                        /* in packets */
                        break;
                default:
                        fprintf(tempfile,"unknown");
                        break;
```

```
                }
                fprintf(tempfile,"\n");
                count++;
        }
        }
fclose(tempfile);
fprintf(stderr,"Number of records: %d\n",count);
if (count >= TCPLLEN){
        perror("Too much data for buffer\n");
        exit(-1);
                }
system("sort /tmp/xxx > /tmp/yyy;cp /tmp/yyy /tmp/xxx;./post_process.sh /tmp/xxx
 /tmp/zzz;cp /tmp/zzz /tmp/xxx");
        exit(0);
}
```

A.2 Test utilities

A.2.1 Program to Send Data (FreeBSD and Linux) - send.c

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <sys/uio.h>
#include <netinet/in.h>

#include <err.h>
#include <errno.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main(int argc, char **argv) {
        char buffer[1448];
        struct addrinfo hints, *res, *res0;
        struct sockaddr name;
        int length;
        int error;
        int s[2];
        char dest[2][3] = { "D1", "D2" }; /* Set in hosts file */
        const char *cause = NULL;
        int blocks, i, j;
        struct timeval before, after;
        double elapsed;
        struct sockaddr_in src[2];

        if (argc != 3) {
                fprintf(stderr, "usage: %s address blocks", argv[0]);
                exit(1);
        }
        src[0].sin_family =  AF_INET;
        src[0].sin_addr.s_addr = inet_addr("192.168.1.51"); /* Local address */
        src[0].sin_port = INADDR_ANY;
        src[1].sin_family =  AF_INET;
        src[1].sin_addr.s_addr = inet_addr("192.168.2.51"); /* Local address */
```

```
src[1].sin_port = INADDR_ANY;


for (j=0;j<2;j++) {     /* number of flows */
/*
 * Open a connection.
 */
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
/*error = getaddrinfo(argv[1], "discard", &hints, &res0);*/
error = getaddrinfo(dest[j], "discard", &hints, &res0);
if (error) {
        errx(1, "%s", gai_strerror(error));
        /*NOTREACHED*/
}
s[j] = -1;
cause = "no addresses";
errno = EADDRNOTAVAIL;
for (res = res0; res; res = res->ai_next) {
        s[j] = socket(res->ai_family, res->ai_socktype,
            res->ai_protocol);
        if (s[j] < 0) {
                cause = "socket";
                continue;
        }
        bind(s[j],(struct sockaddr *)&src[j],sizeof(struct sockaddr));
        if (connect(s[j], res->ai_addr, res->ai_addrlen) < 0) {
                cause = "connect";
                close(s[j]);
                s[j] = -1;
                continue;
        }
        break;  /* okay we got one */
}
if (s[j] < 0) {
        err(1, cause);
        /*NOTREACHED*/
}
freeaddrinfo(res0);
} /* End for */

/*
```

```
         * Read number of blocks.
         */
        blocks = atoi(argv[2]);
        if (blocks <= 0) {
                fprintf(stderr, "blocks must be >= 0, not %d", blocks);
                exit(1);
        }

        gettimeofday(&before, NULL);
        for (i = 0; i < blocks; i++) {
        for (j=0;j<2;j++) {     /* number of flows */
                if (write(s[j], buffer, sizeof(buffer)) <= 0) {
                        perror("write failed");
                        exit(1);
                }
        }
        }

        gettimeofday(&after, NULL);
        length = sizeof(name);
        if (getsockname(s[0], &name, &length)) {
                perror("getting socket name");
                }
        elapsed = after.tv_sec - before.tv_sec +
            1E-6*(after.tv_usec - before.tv_usec);
        printf("port %d, %d MSS blocks, %f seconds, %f Bps, %f bps\n", name.sa_d
ata[0]*256 + name.sa_data[1], blocks, elapsed, 1024.0*blocks/elapsed, 8192.0*blo
cks/elapsed);
        close(s[0]);
        close(s[1]);

        exit(0);
}
```

A.2.2 Program to Extract Data from Memory Dump - Awk Code

```
#!/bin/sh
awk -v port=$2 '$2==3 && ($4==0 || $4 == port) {print}' $1 | sort | awk '
BEGIN{start=0;}
```

```
{
if (NR == 1)
start = $1;
}
END{printf start}
```

A.3 NS Simulations

A.3.1 Generic NS Simulation Script - OTcl Code

```
# Generic Simulation script.
#
# Robert Kilduff    Hamilton Institute  23/04/03.
#
remove-all-packet-headers
add-packet-header IP TCP
set endtime 10.0
set numflows 2
set delay 20ms
set qmax 17

set ns [new Simulator]

#Open the output files
set f [open out.tr w]
for {set i 0} {$i < [expr $numflows]} {incr i} {
set f$i [open out$i.tr w]
puts [set f$i] "\" Flow $i cwnd(pkts)\""
}
puts $f0 "TitleText: Reno flows with constant delay"
set q [open q.tr w]
set nf [open out.nam w]
$ns trace-all $f
$ns namtrace-all $nf

proc finish {} {
    global numflows
    for {set i 0} {$i < [expr $numflows]} {incr i} {
        global f$i
    close [set f$i]
    }
        global ns nf q
    $ns flush-trace
    close $nf
    close $q
        exit 0
}
```

```
proc record {} {
    global numflows
    for {set i 0} {$i < [expr $numflows]} {incr i} {
        global f$i
        global tcp_src$i
    }
    global qmon q
        #Get an instance of the simulator
        set ns [Simulator instance]
        #Set the time after which the procedure should be called again
        set time 0.0001
        #Get the current time
        set now [$ns now]
    for {set i 0} {$i < [expr $numflows]} {incr i} {
        puts [set f$i] "$now [[set tcp_src$i] set cwnd_] [[set tcp_src$i] set ssthr
    }
    puts $q "$now [$qmon set pkts_]"
    #Re-schedule the procedure
        $ns at [expr $now+$time] "record"
}

Agent/TCP set window_ 2000
Agent/TCP set packetSize_ 1460
#Agent/TCP set max_ssthresh_ 10000

set node0 [$ns node]
set node1 [$ns node]
set node2 [$ns node]
set node3 [$ns node]
$ns duplex-link $node0 $node2 100Mb 0ms DropTail
$ns duplex-link $node1 $node2 100Mb 0ms DropTail
$ns duplex-link $node2 $node3 10Mb $delay DropTail
$ns queue-limit $node2 $node3 $qmax
set qmon [$ns monitor-queue $node2 $node3 $q 0.0001]

for {set i 0} {$i < $numflows} {incr i} {
set tcp_src$i [new Agent/TCP/Newreno]
set tcp_snk$i [new Agent/TCPSink]
$ns attach-agent [set node[expr $i % 2]] [set tcp_src$i]
puts "node[expr $i % 2]"
$ns attach-agent $node3 [set tcp_snk$i]
$ns connect [set tcp_src$i] [set tcp_snk$i]
```

```
set ftp$i [new Application/FTP]
[set ftp$i] attach-agent [set tcp_src$i]
puts "ftp$i"
}

#Start logging the received bandwidth
$ns at 0.0 "record"
for {set i 0} {$i < $numflows} {incr i} {
$ns at 0.0 "[set ftp$i] start"
}
#$ns at 0.0 "$ftp0 start"
$ns at 40.0 "$ftp0 stop"
$ns at $endtime  "finish"

#Run the simulation
$ns run
```

A.3.2 Non Synchronised Flows - Dummynet Configuration

```
ipfw add 400 pipe 1 tcp from any to any 9 out via em0
ipfw pipe 1 config bw 800Kbit/s delay 100ms queue 15
ipfw add 401 pipe 2 tcp from any 9 to any in via em0
ipfw pipe 2 config bw 800Kbit/s delay 100ms queue 0

ipfw add 402 pipe 3 tcp from any to any 9 in via em1
ipfw pipe 3 config bw 8000Kbit/s delay 5ms queue 0
ipfw add 403 pipe 4 tcp from 192.168.3.51 to 192.168.1.51 out via em1
ipfw pipe 4 config bw 8000Kbit/s delay 5ms queue 0

ipfw add 404 pipe 5 tcp from any to any 9 in via em2
ipfw pipe 5 config bw 8000Kbit/s delay 25ms queue 0
#ipfw pipe 5 config bw 8000Kbit/s delay 4ms queue 0
ipfw add 405 pipe 6 tcp from 192.168.3.51 to 192.168.2.51 out via em2
ipfw pipe 6 config bw 8000Kbit/s delay 25ms queue 0
#ipfw pipe 6 config bw 8000Kbit/s delay 4ms queue 0
```

A.3.3 Non Synchronised Flows - OTcl script

```
# EXP 8. To look at non synchronised flows
#
# Robert Kilduff        Hamilton Institute      23/10/03.
#
remove-all-packet-headers
add-packet-header IP TCP
set endtime 80.0
set numflows 2
set delay 100ms
set qmax 15

set ns [new Simulator]

#Open the output files
set f [open out w]
for {set i 0} {$i < [expr $numflows]} {incr i} {
set f$i [open out$i w]
}

proc finish {} {
        global numflows
        for {set i 0} {$i < [expr $numflows]} {incr i} {
        global f$i
        close [set f$i]
        }
        global ns nf q
        $ns flush-trace
        close $nf
        close $q
        for {set i 0} {$i < [expr $numflows]} {incr i} {
        global f$i
        global tcp_src$i
        }
        global qmon q
        #Get an instancef0 f1 f2 qmon
        #Get an instance of the simulator
        set ns [Simulator instance]
        #Set the time after which the procedure should be called again
        set time 0.01
        #Get the current time
```

```
        set now [$ns now]
        for {set i 0} {$i < [expr $numflows]} {incr i} {
        }
        puts $q "$now [$qmon set pkts_]"
        #Re-schedule the procedure
        $ns at [expr $now+$time] "record"
}

Agent/TCP set window_ 2000
Agent/TCP set packetSize_ 960

set node0 [$ns node]
set node1 [$ns node]
set node2 [$ns node]
set node3 [$ns node]
$ns duplex-link $node0 $node2 8Mb 5ms DropTail
$ns duplex-link $node1 $node2 8Mb 25ms DropTail
$ns duplex-link $node2 $node3 800Kb $delay DropTail
$ns queue-limit $node2 $node3 $qmax
set qmon [$ns monitor-queue $node2 $node3 $q 0.01]

for {set i 0} {$i < $numflows} {incr i} {
set tcp_src$i [new Agent/TCP/Sack1]
set tcp_snk$i [new Agent/TCPSink/Sack1]
$ns attach-agent [set node[expr $i % 2]] [set tcp_src$i]
puts "node[expr $i % 2]"
$ns attach-agent $node3 [set tcp_snk$i]
$ns connect [set tcp_src$i] [set tcp_snk$i]
set ftp$i [new Application/FTP]
[set ftp$i] attach-agent [set tcp_src$i]
puts "ftp$i"
}

#Start logging the received bandwidth
$ns at 0.0 "record"
for {set i 0} {$i < $numflows} {incr i} {
$ns at 0.0 "[set ftp$i] start"
}
$ns at 40 "$ftp1 stop"
$ns at $endtime  "finish"

#Run the simulation
$ns run
```

A.4 Matlab Simulations

A.4.1 Hybrid Model Source Code - Matlab Code

```
% Implementation of Hybrid Systems model of TCP Newreno
% congestion control.
%
% Robert Kilduff    31/07/03
% Hamilton Institute

function xdot = hybrid_sim4()
clear all;
global B;
B = 10000000/(1500*8); % 10Mb for 12000 bit packets
Window_gain = 1;
global number_of_flows;
number_of_flows = 1;
global RTTF;
RTTF = 0.040;
global RTT;
RTT = [];
RTT(1:number_of_flows,1) = RTTF;
DDD = RTTF;
Tf = 10;    % Finish time
T0 = 0;
global STEP;
STEP=0.001;
time = [0];
timer0 = zeros(number_of_flows,1);
timer = [];
timer(:,1)=timer0;
tdot = [];
tdot = timer0;
w0 = 1*ones(number_of_flows,1);
global w;
w = [];
w(:,1)=w0;
% r for the sender
rw0 = 0*[1:number_of_flows]';
global rw;
```

```
rw = [];
rw(:,1) = rw0;
% loop through all time-steps;
global count;
count = 1;
filter_count = 0;
% Causes variation in the rate of rise of cwnd. Setting it to beta= 1.6
% seems get a better a better result for case where B=833.33, RTTF = 0.040
% and queue = 17. Thus link + queue capacity is 50 packets and link should
% overflow then.
%beta = 1.57188539;
beta = 1.45
L = 1;
Wth = [];
Wth0 = 2000*ones(number_of_flows,1);
Wth(:,1) = Wth0;
k = [];
k0 = 0*ones(number_of_flows,1);
k(:,1) = k0;
m = 2;
% state can be SS = Slow Start, SSD = Slow Start Delay, FR = Fast Recovery,
% TO = TimeOut, CA = Congestion Avoidance, CAD = Congestion Avoidance Delay
for i = 1:number_of_flows state{i} = 'SS'; end
%Queue stuff
% qstate can be QE = Queue Empty, QNF = Queue Not Full, QF = Queue Full
global qstate;
qstate = 'QE';
% states [q1,...qn]
q0 = 0*[1:(number_of_flows)]';
global q;
q = [];
q(:,1)=q0;
% states [r1,..rn]
rq0 = 0*[1:(number_of_flows)]';
global rq;
rq = [];
rq(:,1)=rq0;
global drops;
drops = [];
drops0 = 0*[1:(number_of_flows)]';
drops(:,1) = drops0;
global dropstime;
dropstime = [];
```

```
global zl;
zl = [];
zl(1) =0;


for i = T0:STEP:Tf;

for j = 1:number_of_flows
    state
 switch state{j}
        case 'SS'
            timer(j,count+1) = 0;
            wdot = (log(m)/RTT(j,count))*w(j,count);
            w(j,count+1) = w(j,count)+wdot*STEP; %Euler
            rw(j,count+1) = (beta*w(j,count)/RTT(j,count));
            % Check for Congestion Avoidance
            if w(j,count+1) >= Wth(j);
                w(j,count+1) = Wth(j);
                state{j} = 'CA';
            end
            % Check for Slow Start Delay
            if drops(j) > 0
                timer(j,count+1) = RTT(j,count);
                state{j} = 'SSD';
            end;
            queue(j);
        case 'SSD'
            tdot = -1;
            wdot = (log(m)/RTT(j,count))*w(j,count);
            w(j,count+1) = w(j,count)+wdot*STEP; %Euler
            timer(j,count+1) = timer(j,count)+tdot*STEP; %Euler
            rw(j,count+1) = (beta*w(j,count)/RTT(j,count));
            %check for Fast Recovery
            if timer(j,count+1) < 0;
                state{j} = 'FR';
                if drops(j) > w(j,count+1)/2 + 1
                    k(j) = 1+ceil(log2(drops(j)));
                else
                    k(j) = ceil(log2((1+(w(j,count)/2))/(1+(w(j,count)/2)
                    -drops(j))));
                end
                rw(j,count+1) = (1+(w(1,count)/2)-drops(j))/RTT(j,count);
                w(j,count+1) = floor(w(j,count)/2);
```

```
        timer(j,count+1) = RTT(j,count);
    elseif w(j,count+1) < max(2+drops(j), 2*drops(j)-4)
        timer(j,count+1) = 1;
        w(j,count+1) = floor(w(j,count+1)/2);
        state{j} = 'TO';
    end
    queue(j);
case 'FR'
    tdot = -1;
    wdot = 0;
    rwdot = 0;
    w(j,count+1)=w(j,count)+wdot*STEP; %Euler
    rw(j,count+1) = rw(j,count)+rwdot*STEP; %Euler
    timer(j,count+1) = timer(j,count)+tdot*STEP; %Euler
    % Check Congestion Avoidance
    if timer(j,count+1) < 0 & k(j) > 0
       timer(j,count+1) = RTT(j,count);
       k(j) = k(j) - 1;
       if k(j) <= 0
        state{j} = 'CA';
        drops(j) = 0;
       end
       rw(j,count+1) = 2*rw(j,count+1);
   end
   queue(j);
case 'TO'
    tdot = -1;
    wdot = 0;
    rwdot = 0;
    w(j,count+1)=w(j,count)+wdot*STEP;
    rw(j,count+1)=0;
    timer(j,count+1) = timer(j,count)+tdot*STEP; %Euler
    % Check for Slow Start
    if timer(j,count+1) < 0
        state{j} = 'SS';
        Wth(j) = w(j,count+1)/2;
        w(j,count+1)=1;
    end;
    queue(j);
case 'CA'
    timer(j,count+1) = 0;
    wdot = (L/(RTT(j,count)));
    w(j,count+1)=w(j,count)+wdot*STEP;
```

```
            rw(j,count+1)=w(j,count)/RTT(j,count);
            if drops(j) > 0
                timer(j,count+1) = RTT(j,count);
                state{j} = 'CAD';
            end
            queue(j);
        case 'CAD'
            tdot = -1;
            wdot = (L/RTT(j,count));
            w(j,count+1)=w(j,count)+wdot*STEP;
            rw(j,count+1)=w(j,count)/RTT(j,count);
            timer(j,count+1)=timer(j,count)+tdot*STEP;
            %check for Fast Recovery
            if timer(j,count+1) < 0;
                state{j} = 'FR';
                if drops(j) > w(j,count+1)/2 + 1
                    k(j) = 1+ceil(log2(drops(j)));
                else
                    k(j) = ceil(log2((1+(w(j,count)/2))/(1+(w(j,count)/2)
                    -drops(j))));
                end
                rw(j,count+1) = (1+(w(j,count)/2)-drops(j))/RTT(j,count);
                w(j,count+1) = floor(w(j,count)/2);
                timer(j,count+1) = RTT(j,count);
            end
            %check for TimeOut
            if w(j,count+1) < max (2+drops(j), 2*drops(j)-4)
                timer(j,count+1) = 1;
                w(j,count+1) = floor(w(j,count)/2);
                state{j} = 'TO';
            end
            queue(j);
    end;
end;
count = count + 1;
end;

output;

function queue(j)
global number_of_flows;
global B;
global count;
```

```
global q;
persistent ql;
global rq;
global w;
global rw;
global drops;
global STEP;
global qstate;
global zl;
global zlcount;
global dropstime;
global RTT;
global RTTF;
Q_max = 17 - 1; % Queue size in ns is number of packets queue can hold +1

sl = sum(rw(1:j,count+1),1) + sum(rw(j+1:number_of_flows,count),1);

switch qstate
    case 'QE'
        if  (sl) <= B;
            rq(j,count+1) = rw(j,count+1);
        else
            rq(j,count+1) = (rw(j,count)/sl)*B;
        end
        qdot = rw(j,count+1) - rq(j,count+1);
        q(j,count+1)=q(j,count)+qdot*STEP; % Euler
        RTT(j,count+1) = RTTF;
        if sum(q(1:j,count+1),1)+sum(q(j+1:number_of_flows,count),1) > 0
            RTT(j,count+1) = RTTF + q(j,count+1)/B;
            ql = sum(q(1:j,count+1),1)+sum(q(j+1:number_of_flows,count),1);
            qstate = 'QNF';
        end
    case 'QNF'
        ql
        rq(j,count+1) = q(j,count)/ql*B;
        qdot = rw(j,count+1) - rq(j,count+1);
        q(j,count+1)=max(0,q(j,count)+qdot*STEP); % Euler
        RTT(j,count+1) = RTTF + ql/B;
        ql = sum(q(1:j,count+1),1)+sum(q(j+1:number_of_flows,count),1);
        if (ql >= Q_max)
            q(j,count+1) = Q_max - sum(q(1:j-1,count+1),1)
            -sum(q(j+1:number_of_flows,count),1);
        end
```

```
    if (ql >= Q_max) & (sl > B)
        qstate = 'QF';
        ql = Q_max;
        zl(count+1)=0;
        zlcount=0;
        drops(j) = 1;
        dropstime = [dropstime,count]; %time of drop
    elseif ql <= 0
        qstate = 'QE';
        RTT(j,count+1) = RTTF;
        q(1:number_of_flows,count+1) = 0;
    end
case 'QF'
    if ((sl)) <= B
        qstate = 'QNF';
        q(1:number_of_flows,count+1) = q(1:number_of_flows,count);
        RTT(j,count+1) = RTTF + ql/B;
        return
    end
    zldot = sl - B;
    qdot = ((rw(j,count+1)/sl) - (q(j,count)/ql))*B;
    rw(j,count+1);
    q(j,count+1)=q(j,count)+qdot*STEP; % Euler
    rq(j,count+1) = q(j,count+1)/ql*B;
    if (ql >= Q_max)
        q(j,count+1) = Q_max - sum(q(1:j-1,count+1),1)
        -sum(q(j+1:number_of_flows,count),1);
    end
    ql = sum(q(1:j,count+1),1)+sum(q(j+1:number_of_flows,count),1)
    RTT(j,count+1) = RTTF + ql/B;
    % Assume when queue full that queue is shared out equally between
    % flows.
    if ql > Q_max
        q(j:number_of_flows,count+1) = Q_max/number_of_flows;
    end
    zl(count+1)=zl(count)+zldot*STEP; % Euler
    if zl(count+1) > 1
        zlcount = zlcount+1;
        % Lets assume round robin distribution
        drops(mod(zlcount,number_of_flows)+1) =
        drops(mod(zlcount,number_of_flows)+1) + 1;
        zl(count+1) = 0;
    end
```

```
end

function output()
global count;
global STEP;
global w;
global rw;
global q;
global RTT;
global RTTF;
global dropstime;
global number_of_flows;
hold off
plot([1:count]*STEP,(w(1,:)),'b')
hold on
plot([1:count]*STEP,(w(2,:)),'r')
plot([1:count]*STEP,(q(1,:)),'m')
if length(dropstime) > 0
    plot(dropstime*STEP,1,'g+');
end
hold off
```

A.4.2 Hybrid Model Discrete Queue - Matlab Code

```
% Implementation of Hybrid Systems model of TCP-Newreno
% congestion control. Discrete Queue.
%
% Robert Kilduff    10/09/03
% Hamilton Institute

function xdot = hybrid_sim6()
clear all;
global B;
B = 10000000/(1500*8); % 10Mb for 12000 bit packets
Window_gain = 1;
global number_of_flows;
number_of_flows = 2;
global RTTF;
RTTF = [];
RTTF(1,1) = 0.040;
RTTF(2,1) = 0.040;
global RTT;
RTT = [];
RTT(1:number_of_flows,1) = RTTF(1:number_of_flows,1);
DDD = RTTF(1);
Tf =10;
T0 = 0;
global STEP;
STEP=1/(10*B);
time = [0];
timer0 = zeros(number_of_flows,1);
timer = [];
timer(:,1)=timer0;
tdot = [];
tdot = timer0;
w0 = 13*ones(number_of_flows,1);
global w;
w = [];
w(:,1)=w0;
rw0 = 0*[1:number_of_flows]';
global rw;
rw = [];
rw(:,1) = rw0;
global count;
```

```
count = 1;
filter_count = 0;
global beta;
beta = 1.45;
L = 1;
Wth = [];
Wth0 = 2000*[1:number_of_flows]';
Wth(:,1) = Wth0;
m = 2;
ndrops = 1; % simple case first
% state can be SS = Slow Start, SSD = Slow Start Delay, FR = Fast Recovery,
% TO = TimeOut, CA = Congestion Avoidance, CAD = Congestion Avoidance Delay
global state;
state = [];
for i = 1:number_of_flows state{i} = 'CA'; end

%Queue stuff
% qstate can be QE = Queue Empty, QNF = Queue Not Full, QF = Queue Full
global qstate;
qstate = 'QNF';
global Q_max;
Q_max = 17 - 1; % Queue size in ns is number of packets queue can hold + 1
q0 = 0*[1:(Q_max)]';
global q;
q = [];
q(:,1)=q0;
global qptr;
qptr = 0;
% states [r1,..rn]
rq0 = 0*[1:(number_of_flows)]';
global rq;
rq = [];
rq(:,1)=rq0;
global drops;
drops = [];
drops0 = 0*[1:(number_of_flows)]';
drops(:,1) = drops0;
global dropstime;
dropstime = [];
global tq;
tq = [];
tq(1) = 1/B;
global te;
```

```
te = [];
te(1) = 0;
global encount;
encount = 1;
global enpacket;
global k;
k = [];
k0 = 0*[1:(number_of_flows)]';
k(:,1) = k0;
enpacket = 0;
global wtmp;
wtmp = [];
wtmp0 = 0*[1:(number_of_flows)]';
wtmp = wtmp0;
global timertmp;
global boundary;
boundary = [];
b0 = 0*[1:(number_of_flows)]';
boundary(:,1) = b0;
global lastw;
lastw = [];
l0 = 1*[1:(number_of_flows)]';
lastw(:,1) = l0;
timecount =1;

for i = T0:STEP:Tf;
if count*STEP > timecount
    count*STEP
    timecount = timecount + 1;
end
for j = 1:number_of_flows
 switch state{j}
        case 'SS'
            timer(j,count+1) = 0;
            wdot = (log(m)/RTT(j,count))*w(j,count);
            w(j,count+1) = w(j,count)+wdot*STEP; %Euler
            rw(j,count+1) = (beta*w(j,count)/RTT(j,count));
            % Check for Congestion Avoidance
            if w(j,count+1) >= Wth(j);
                w(j,count+1) = Wth(j);
                state{j} = 'CA';
            end
            % Check for Slow Start Delay
```

```
    if drops(j) > 0
        %w(number_of_flows+1,count) = DDD;
        timer(j,count+1) = RTT(j,count);
        wtmp(j) = w(j,count+1);
        state{j} = 'SSD';
    end;
    queue(j);
case 'SSD'
    tdot = -1;
    wdot = (log(m)/RTT(j,count))*w(j,count);
    w(j,count+1) = w(j,count)+wdot*STEP; %Euler
    timer(j,count+1) = timer(j,count)+tdot*STEP; %Euler
    rw(j,count+1) = (beta*(w(j,count+1) -drops(j) - drops(j)/(wtmp(j)+1))
    /RTT(j,count));
    if timer(j,count+1) < 0;
        state{j} = 'FR';
        if drops(j) > w(j,count+1)/2 + 1
            k(j) = 1+ceil(log2(drops(j)));
        else
            k(j) = ceil(log2((1+(w(j,count)/2))/(1+(w(j,count)/2)
            -drops(j))));
        end
        rw(j,count+1) = (1+(w(1,count)/2)-drops(j))/RTT(j,count);
        w(j,count+1) = floor(w(j,count)/2);
        timer(j,count+1) = RTT(j,count);
        timertmp=RTT(j,count)*k;
        drops(j) = 0;
    elseif w(j,count+1) < max(2+drops(j), 2*drops(j)-4)
        timer(j,count+1) =1;
        w(j,count+1) = floor(w(j,count+1)/2);
        drops(j) = 0;
        state{j} = 'TO';
    end
    queue(j);
case 'FR'
    tdot = -1;
    wdot = 0;
    rwdot = 0;
    w(j,count+1)=w(j,count)+wdot*STEP; %Euler
    rw(j,count+1) = rw(j,count)+rwdot*STEP; %Euler
    timer(j,count+1) = timer(j,count)+tdot*STEP; %Euler
    % Check Congestion Avoidance
    if timer(j,count+1) < 0 & k(j) > 0
```

```
        timer(j,count+1) = RTT(j,count);
        k(j) = k(j) - 1;
        if k(j) <= 0
         drops(j) = 0;
         state{j} = 'CA';
        end
        rw(j,count+1) = 2*rw(j,count+1);
    end
    queue(j);
case 'TO'
    tdot = -1;
    wdot = 0;
    rwdot = 0;
    w(j,count+1)=w(j,count)+wdot*STEP;
    rw(j,count+1)=0;
    timer(j,count+1) = timer(j,count)+tdot*STEP; %Euler
    % Check for Slow Start
    if timer(j,count+1) < 0
        state{j} = 'SS';
        Wth(j) = w(j,count+1)/2;
        w(j,count+1)=1;
    end;
    queue(j);
case 'CA'
    timer(j,count+1) = 0;
    wdot = (L/(RTT(j,count)));
    w(j,count+1)=w(j,count)+wdot*STEP;
    rw(j,count+1)=w(j,count)/RTT(j,count);
    if drops(j) > 0
        timer(j,count+1) = RTT(j,count);
        wtmp(j) = w(j,count+1);
        state{j} = 'CAD';
    end
    queue(j);
case 'CAD'
    % possibly move next line to check section above
    tdot = -1;
    wdot = (L/RTT(j,count));
    w(j,count+1)=w(j,count)+wdot*STEP;
    % Account for dropped packet by subtracting 1 from cwnd as
    % cwnd no longer represents the amount of packets in flight.
    % Also must subtract 1/cwnd from rate due to the effective
    % reduction of cwnd increase due to loss of packet.
```

```
            % Note this increase occurs at time of packet drop + 1 RTT.
            rw(j,count+1)=(w(j,count) -1 - drops(j)/(wtmp(j)+1))/RTT(j,count);
            timer(j,count+1)=timer(j,count)+tdot*STEP;
            %check for Fast Recovery
            if timer(j,count+1) < 0;
                state{j} = 'FR';
                if drops(j) > w(j,count+1)/2 + 1
                    k(j) = 1+ceil(log2(drops(j)));
                else
                    k(j) = ceil(log2((1+(w(j,count)/2))/(1+(w(j,count)/2)
                    -drops(j))));
                end
                rw(j,count+1) = (1+(w(j,count)/2)-drops(j))/RTT(j,count);
                w(j,count+1) = floor(w(j,count)/2);
                timer(j,count+1) = RTT(j,count);
                drops(j) = 0;
                lastw(j,1) = 1;
            %check for TimeOut
        elseif w(j,count+1) < max (2+drops(j), 2*drops(j)-4)
                timer(j,count+1) = 1;
                w(j,count+1) = floor(w(j,count)/2);
                drops(j) = 0;
                state{j} = 'TO';
            end
            queue(j);
    end;
end;
count = count + 1;
end;

output;

function queue(j)
global number_of_flows;
global B;
global count;
global q;
global rq;
global tq;
global rw;
global drops;
global STEP;
global dropstime;
```

```
global RTT;
global RTTF;
global qptr;
global Q_max;
global state;
global qstate;
global w;
global te;
global encount;
global enpacket;
persistent rqdot;
global lastw;
global boundary;
global wtmp;
global beta;

if strcmp(state(j),'CAD')
    rqdot = (floor(w(j,count+1)) -drops(j) -1
     - drops(j)/(floor(wtmp(j))+1))/RTT(j,count);
else
rqdot = (floor(w(j,count+1)) -1)/RTT(j,count);
end
% Integrate fluid from flows
rq(j,count+1) = rq(j,count) + rqdot*STEP;

if (j==1)
q(:,count+1) = q(:,count);
te(count+1) = te(count);
tq(count+1) = tq(count);

% Service the Queue
if qptr > 0
    tqdot = -1;
    tq(count+1) = tq(count) + tqdot*STEP;
    if tq(count+1) <= 0
        q(1:min(qptr,Q_max-1), count+1) = q(2:min(qptr+1,Q_max),count+1);
        q(min(qptr+1,Q_max):Q_max, count+1) = 0;
        qptr = qptr -1;
        tq(count+1) = 1/B;
    end
    if qptr < Q_max
        qstate = 'QNF';
    end
```

```
end
end

if w(j,count+1) >= lastw(j)
    boundary (j)= 1;
    lastw(j) = floor(w(j,count+1))+1;
end

if qptr < Q_max && boundary(j) == 1
    if qptr < Q_max - 1
        qptr = qptr + 2;
        q(qptr-1:qptr,count+1) = j;
        rq(j,count+1) = rq(j,count+1) - 1;
    else % qptr = Q_max - 1
        qptr = qptr + 1;
        q(qptr,count+1) = j;
        drops(j) = drops(j) + 1;
        dropstime = [dropstime,[count*STEP,j]'];
        rq(j,count+1) = rq(j,count+1) -1;
    end
    boundary(j) = 0;
elseif qptr == Q_max && boundary(j) == 1
    drops(j) = drops(j) + 2;
    dropstime = [dropstime,[count*STEP,j]'];
    rq(j,count+1) = rq(j,count+1) - 1;
    boundary(j) = 0;
end



if rq(j,count) >= 1
    if qptr < Q_max
            qptr = qptr + 1;
            q(qptr,count+1) = j;
    else
        drops(j) = drops(j) + floor(rq(j,count+1));
        dropstime = [dropstime,[count*STEP,j]'];
    end
        rq(j,count+1) = rq(j,count+1) - 1;
end

RTT(j,count+1) = RTTF(j) + qptr/B;
```

```
% End of function Queue

function output()
global count;
global STEP;
global w;
global rw;
global q;
global rq;
global RTT;
global RTTF;
global dropstime;
global number_of_flows;
global te;
hold off
plot([1:count]*STEP,(w(1,:)),'b')
hold on
plot([1:count]*STEP,(w(2,:)),'c')
plot([1:count]*STEP,sum(q(:,1:count)~=0),'m')
plot([1:count]*STEP,te(1:count)*1000,'k');
plot([1:count]*STEP,RTT(1,1:count)*1000,'g');
if length(dropstime) > 0
    for i = 1:length(dropstime)
        switch round(dropstime(2,i))
            case 1
                plot(dropstime(1,i),14,'b+');
            case 2
                plot(dropstime(1,i),14,'c+');
        end
    end
end
hold off
```

A.4.3 Hybrid Model Discrete Queue with Back-to-back Packets - Matlab Code

```
% Implementation of Hybrid Systems model of TCP-Newreno
% congestion control. Discrete Queue.
%
% Robert Kilduff    29/08/03
% Hamilton Institute

function hybrid_dis
global B;
B = 10000000/(1500*8); % 10Mb for 12000 bit packets
Window_gain = 1;
global number_of_flows;
number_of_flows = 2;
global RTTF;
RTTF = [];
RTTF(1,1) = 0.040;
RTTF(2,1) = 0.040;
global RTT;
RTT = [];
RTT(1:number_of_flows,1) = RTTF(1:number_of_flows,1);
DDD = RTTF(1);
Tf =10;
%Tf = 1.2329;
T0 = 0;
global STEP;
STEP=1/(3*B);;
time = [0];
timer0 = zeros(number_of_flows,1);
timer = [];
timer(:,1)=timer0;
tdot = [];
tdot = timer0;
w0 = 13*ones(number_of_flows,1);
global w;
w = [];
w(:,1)=[13; 12];
rw0 = 0*[1:number_of_flows]';
global rw;
rw = [];
rw(:,1) = rw0;
```

```
global count;
count = 1;
filter_count = 0;
global beta;
beta = 1.45;
L = 1;
Wth = [];
Wth0 = 2000*[1:number_of_flows]';
Wth(:,1) = Wth0;
m = 2;
ndrops = 1; % simple case first
% state can be SS = Slow Start, SSD = Slow Start Delay, FR = Fast Recovery,
% TO = TimeOut, CA = Congestion Avoidance, CAD = Congestion Avoidance Delay
global state;
state = [];
for i = 1:number_of_flows state{i} = 'CA'; end

%Queue stuff
global Q_max;
Q_max = 17 - 1; % Queue size in ns is number of packets queue can hold + 1
q0 = 0*[1:(Q_max)]';
global q;
q = [];
q(:,1)=q0;
global qptr;
qptr = 0;
% states [r1,..rn]
rq0 = 0*[1:(number_of_flows)]';
global rq;
rq = [];
rq(:,1)=rq0;
global drops;
drops = [];
drops0 = 0*[1:(number_of_flows)]';
drops(:,1) = drops0;
global dropstime;
dropstime = [];
global tq;
tq = [];
tq(1) = 1/B;
global k;
k = [];
k0 = 0*[1:(number_of_flows)]';
```

```
k(:,1) = k0;
global wtmp;
wtmp = zeros(1,number_of_flows);
global timertmp;
global boundary;
boundary = zeros(1,number_of_flows);
global rq2;
rq2=zeros(1,number_of_flows);
timecount =1;
startj=0;

w(1,1)=13; w(2,1)=12; qptr=1; q(1:qptr,1)=zeros(qptr,1);
for i = T0:STEP:Tf;
    if count*STEP >= timecount
        count*STEP
        timecount = timecount + 1;
    end

    servicequeue;
    for j=1:number_of_flows
       RTT(j,count+1) = RTTF(j) + (qptr)/B;
    end

    startj=startj+1;
    for jj = 1:number_of_flows
    j=jj;
       switch state{j}
            case 'SS'
                timer(j,count+1) = 0;
                wdot = (log(m)/RTT(j,count))*w(j,count);
                w(j,count+1) = w(j,count)+wdot*STEP; %Euler
                rw(j,count+1) = (beta*w(j,count)/RTT(j,count));
                % Check for Congestion Avoidance
                if w(j,count+1) >= Wth(j);
                    w(j,count+1) = Wth(j);
                    state{j} = 'CA';
                end
                % Check for Slow Start Delay
                if drops(j) > 0
                    timer(j,count+1) = RTT(j,count);
                    wtmp(j) = w(j,count+1);
                    state{j} = 'SSD';
                end;
```

```
    queue(j);
case 'SSD'
    tdot = -1;
    wdot = (log(m)/RTT(j,count))*w(j,count);
    w(j,count+1) = w(j,count)+wdot*STEP; %Euler
    timer(j,count+1) = timer(j,count)+tdot*STEP; %Euler
    rw(j,count+1) = (beta*(w(j,count+1) -drops(j)
    - drops(j)/(wtmp(j)+1))/RTT(j,count));
    if timer(j,count+1) <= 0;
        state{j} = 'FR';
        if drops(j) > w(j,count+1)/2 + 1
            k(j) = 1+ceil(log2(drops(j)));
        else
            k(j) = ceil(log2((1+(w(j,count)/2))/(1+(w(j,count)/2)
            -drops(j))));
        end
        rw(j,count+1) = (1+(w(1,count)/2)-drops(j))/RTT(j,count);
        w(j,count+1) = floor(w(j,count)/2); lastw(j) =
        floor(w(j,count+1))+1;
        timer(j,count+1) = RTT(j,count);
        timertmp=RTT(j,count)*k;
        drops(j) = 0;
    elseif w(j,count+1) < max(2+drops(j), 2*drops(j)-4)
        timer(j,count+1) =1;
        w(j,count+1) = floor(w(j,count+1)/2);
        drops(j) = 0;
        state{j} = 'TO';
    end
    queue(j);
case 'FR'
    tdot = -1;
    wdot = 0;
    rwdot = 0;
    w(j,count+1)=w(j,count)+wdot*STEP; %Euler
    rw(j,count+1) = floor(w(j,count)-1)/RTT(j,count);
    timer(j,count+1) = timer(j,count)+tdot*STEP; %Euler
    if timer(j,count+1) <= 0
       k(j) = k(j) - 1;
       if k(j) <= 0
          drops(j) = 0;
          state{j} = 'CA';
       else
          timer(j,count+1) = RTT(j,count);
```

```
        end
        rw(j,count+1) = 2*rw(j,count+1);
    end
   queue(j);
case 'TO'
    tdot = -1;
    w(j,count+1)=w(j,count);
    rw(j,count+1)=0;
    timer(j,count+1) = timer(j,count)+tdot*STEP; %Euler
    % Check for Slow Start
    if timer(j,count+1) <= 0
        state{j} = 'SS';
        Wth(j) = w(j,count+1)/2;
        w(j,count+1)=1; lastw(j) = floor(w(j,count+1))+1;
    end;
    queue(j);
case 'CA'
    timer(j,count+1) = 0;
    wdot = (L/(RTT(j,count)));
    w(j,count+1)=w(j,count)+wdot*STEP;
    if (rq2(j)>=floor(w(j,count+1))-1)
        rq2(j)=rq2(j)-floor(w(j,count+1)); boundary(j)=1;
    end
    rw(j,count+1)=floor(w(j,count)-1)/RTT(j,count);
    queue(j);
    if drops(j) > 0
        timer(j,count+1) = RTT(j,count);
        wtmp(j) = w(j,count+1);
        state{j} = 'CAD';
    end
case 'CAD'
    tdot = -1;
    wdot = (L/RTT(j,count));
    w(j,count+1)=w(j,count)+wdot*STEP;
    if (rq2(j)>=floor(w(j,count+1))-1)
        rq2(j)=rq2(j)-floor(w(j,count+1)); boundary(j)=1;
    end
    rw(j,count+1)=(w(j,count) -1
    -drops(j)/(wtmp(j)+1))/RTT(j,count);
    timer(j,count+1)=timer(j,count)+tdot*STEP;
    %check for Fast Recovery
    if timer(j,count+1) <= 0;
        state{j} = 'FR';
```

```
                    if drops(j) > w(j,count+1)/2 + 1
                        k(j) = 1+ceil(log2(drops(j)));
                    else
                        k(j) = ceil(log2((1+(w(j,count)/2))
                        /(1+(w(j,count)/2)-drops(j))));
                    end
                    rw(j,count+1) = (1+(w(j,count)/2)-drops(j))/RTT(j,count);
                    w(j,count+1) = floor(w(j,count)/2); lastw(j) =
                    floor(w(j,count+1))+1;
                    timer(j,count+1) = RTT(j,count);
                    drops(j) = 0;
                %check for TimeOut
                elseif w(j,count+1) < max (2+drops(j), 2*drops(j)-4)
                    timer(j,count+1) = 1;
                    w(j,count+1) = floor(w(j,count)/2); lastw(j) =
                    floor(w(j,count+1))+1;
                    drops(j) = 0;
                    state{j} = 'TO';
                end
                queue(j);
        end;
    end;
    count = count + 1;
end;

output;


function servicequeue
global B count STEP q tq qptr Q_max;

q(:,count+1) = q(:,count);
tq(count+1) = tq(count);
% Service the Queue
if qptr > 0
    tqdot = -1;
    tq(count+1) = tq(count) + tqdot*STEP;
    if tq(count+1) <= 0
        q(1:min(qptr,Q_max-1), count+1) = q(2:min(qptr+1,Q_max),count+1);
        q(min(qptr+1,Q_max):Q_max, count+1) = 0;
        qptr = qptr -1;
        tq(count+1) = 1/B;
    end
```

```
end


function queue(j)
global count STEP q qptr Q_max rq rq2 rw drops dropstime boundary;

rq(j,count+1) = rq(j,count) + rw(j, count+1)*STEP;

if (boundary(j)==1)
    added=min(2, Q_max-qptr);
    if qptr < Q_max
        q(qptr+1:qptr+added,count+1) = j;
        qptr = qptr + added;
    end
    if (added < 2)
        drops(j) = drops(j) + 2-added;
        dropstime = [dropstime,[count*STEP,j]'];
    end
    rq(j,count+1) = rq(j,count+1) - 1;
    rq2(j)=rq2(j)+2;
    boundary(j) = 0;
end

if rq(j,count+1) >= 1
    added=min(floor(rq(j,count+1)), Q_max-qptr);
    if qptr < Q_max
        q(qptr+1:qptr+added,count+1) = j;
        qptr = qptr + added;
    end
    rq(j,count+1)=rq(j,count+1)-added;
    rq2(j)=rq2(j)+added;
end

% End of function Queue

function output()
global count;
global STEP;
global w;
global rw;
global q;
global rq;
global RTT;
```

```
global RTTF;
global drops;
global dropstime;
global number_of_flows;
hold off
plot([1:count]*STEP,(w(1,:)),'b')
hold on
plot([1:count]*STEP,(w(2,:)),'c')
plot([1:count]*STEP,sum(q(:,1:count)~=0),'m')
plot([1:count]*STEP,RTT(1,1:count)*1000,'g');
if length(dropstime) > 0
    for i = 1:length(dropstime)
        switch round(dropstime(2,i))
            case 1
                plot(dropstime(1,i),14,'b+');
            case 2
                plot(dropstime(1,i),14,'c+');
        end
    end
end
hold off
```

A.4.4 Hybrid Model Discrete Queue with Entrainment - Matlab Code

```
% Implementation of Hybrid Systems model of TCP Newreno
% congestion control. Discrete Queue with entrainment.
%
% Robert Kilduff    20/09/03
% Hamilton Institute

function xdot = hybrid_en()
clear all;
global B;
B = 10000000/(1500*8); % 10Mb for 12000 bit packets
Window_gain = 1;
global number_of_flows;
number_of_flows = 1;
global RTTF;
RTTF = [];
RTTF(1,1) = 0.040;
RTTF(2,1) = 0.060;
global RTT;
RTT = [];
RTT(1:number_of_flows,1) = RTTF(1:number_of_flows,1);
DDD = RTTF(1);
Tf = 0.35;
T0 = 0;
global STEP;
STEP=1/(10*B);
time = [0];
timer0 = zeros(number_of_flows,1);
timer = [];
timer(:,1)=timer0;
tdot = [];
tdot = timer0;
w0 = 1*ones(number_of_flows,1);
global w;
w = [];
w(:,1)=w0;
rw0 = 0*[1:number_of_flows]';
global rw;
rw = [];
rw(:,1) = rw0;
```

```
global count;
count = 1;
filter_count = 0;
beta = 1.45;
L = 1;
Wth = [];
Wth0 = 2000*[ones(number_of_flows,1)];
Wth(:,1) = Wth0;
m = 2;
ndrops = 1; % simple case first
% state can be SS = Slow Start, SSD = Slow Start Delay, FR = Fast Recovery,
% TO = TimeOut, CA = Congestion Avoidance, CAD = Congestion Avoidance Delay
global state;
state = [];
for i = 1:number_of_flows state{i} = 'SS'; end

%Queue stuff
% qstate can be QE = Queue Empty, QNF = Queue Not Full, QF = Queue Full
global qstate;
qstate = 'QNF';
global Q_max;
Q_max = 17 - 1; % Queue size in ns is number of packets queue can hold + 1
q0 = 0*[1:(Q_max)]';
global q;
q = [];
q(:,1)=q0;
global qptr;
qptr = 0;
% states [r1,..rn]
rq0 = 0*[1:(number_of_flows)]';
global rq;
rq = [];
rq(:,1)=rq0;
global drops;
drops = [];
drops0 = 0*[1:(number_of_flows)]';
drops(:,1) = drops0;
global dropstime;
dropstime = [];
global tq;
tq = [];
tq(1) = 1/B;
global ts;
```

```
ts = [];
ts(1) = 1/B;
global k;
k = [];
k0 = 0*[1:(number_of_flows)]';
k(:,1) = k0;
enpacket = 0;
wtmp = 0;
global timertmp;
global boundary;
boundary = [];
b0 = zeros(number_of_flows,1);
boundary(:,1) = b0;
global lastw;
lastw = [];
lastw(:,1) = ones(number_of_flows,1);
timecount =1;
global lock;
lock = 0;
global pktcnt;
pktcnt = 0;
global encount;
encount = [];
encount(:,1) = ones(number_of_flows,1);
global enmax;
enmax = [];
enmax0 = zeros(number_of_flows,1);
enmax(:,1) = enmax0;


for i = T0:STEP:Tf;


for j = 1:number_of_flows
 switch state{j}
        case 'SS'
            timer(j,count+1) = 0;
            wdot = (log(m)/RTT(j,count))*w(j,count);
            w(j,count+1) = w(j,count)+wdot*STEP; %Euler
            rw(j,count+1) = (beta*w(j,count)/RTT(j,count));
            % Check for Congestion Avoidance
            if w(j,count+1) >= Wth(j);
                w(j,count+1) = Wth(j);
```

```
        state{j} = 'CA';
    end
    % Check for Slow Start Delay
    if drops(j) > 0 ;
        timer(j,count+1) = RTT(j,count);
        wtmp = w(j,count+1);
        state{j} = 'SSD';
    end;
    queue(j);
case 'SSD'
    tdot = -1;
    wdot = (log(m)/RTT(j,count))*w(j,count);
    w(j,count+1) = w(j,count)+wdot*STEP; %Euler
    timer(j,count+1) = timer(j,count)+tdot*STEP; %Euler
    rw(j,count+1) = (beta*(w(j,count+1) -drops(j)
    - drops(j)/(wtmp+1))/RTT(j,count));
    if timer(j,count+1) < 0;
        state{j} = 'FR';
        if drops(j) > w(j,count+1)/2 + 1
            k(j) = 1+ceil(log2(drops(j)));
        else
            k(j) = ceil(log2((1+(w(j,count)/2))/(1+(w(j,count)/2)
            -drops(j))));
        end
        rw(j,count+1) = (1+(w(1,count)/2)-drops(j))/RTT(j,count);
        w(j,count+1) = floor(w(j,count)/2);
        timer(j,count+1) = RTT(j,count);
        timertmp=RTT(j,count)*k;
        drops(j) = 0;
    elseif w(j,count+1) < max(2+drops(j), 2*drops(j)-4)
        % Hybrid model uses 1 second here
        timer(j,count+1) = 1;
        w(j,count+1) = floor(w(j,count+1)/2);
        drops(j) = 0;
        state{j} = 'TO';
    end
    queue(j);
case 'FR'
    tdot = -1;
    wdot = 0;
    rwdot = 0;
    w(j,count+1)=w(j,count)+wdot*STEP; %Euler
    rw(j,count+1) = rw(j,count)+rwdot*STEP; %Euler
```

```
    timer(j,count+1) = timer(j,count)+tdot*STEP; %Euler
    if timer(j,count+1) < 0 & k(j) > 0
       timer(j,count+1) = RTT(j,count);
       k(j) = k(j) - 1;
       if k(j) <= 0
        drops(j) = 0;
        state{j} = 'CA';
       end
       rw(j,count+1) = 2*rw(j,count+1);
   end
   queue(j);
case 'TO'
    tdot = -1;
    wdot = 0;
    rwdot = 0;
    w(j,count+1)=w(j,count)+wdot*STEP;
    rw(j,count+1)=0;
    timer(j,count+1) = timer(j,count)+tdot*STEP; %Euler
    % Check for Slow Start
    if timer(j,count+1) < 0
        state{j} = 'SS';
        Wth(j) = w(j,count+1)/2;
        w(j,count+1)=1;
    end;
    queue(j);
case 'CA'
    timer(j,count+1) = 0;
    wdot = (L/(RTT(j,count)-1/B));
    w(j,count+1)=w(j,count)+wdot*STEP;
    rw(j,count+1)=w(j,count)/RTT(j,count);
    if drops(j) > 0
        timer(j,count+1) = RTT(j,count);
        wtmp = w(j,count+1);
        state{j} = 'CAD';
    end
    queue(j);
case 'CAD'
    tdot = -1;
    wdot = ((L- 1/(wtmp+1))/RTT(j,count)-1/B);
    w(j,count+1)=w(j,count)+wdot*STEP;
    % Account for dropped packet by subtracting 1 from cwnd as
    % cwnd no longer represents the amount of packets in flight.
    % Also must subtract 1/cwnd from rate due to the effective
```

```
            % reduction of cwnd increase due to loss of packet.
            % Note this increase occurs at time of packet drop + 1 RTT.
            rw(j,count+1)=(w(j,count) -1 - drops(j)/(wtmp+1))/RTT(j,count);
            timer(j,count+1)=timer(j,count)+tdot*STEP;
            if timer(j,count+1) < 0;
                state{j} = 'FR';
                if drops(j) > w(j,count+1)/2 + 1
                    k(j) = 1+ceil(log2(drops(j)));
                else
                    k(j) = ceil(log2((1+(w(j,count)/2))/(1+(w(j,count)/2)
                    -drops(j))));
                end
                rw(j,count+1) = (1+(w(j,count)/2)-drops(j))/RTT(j,count);
                w(j,count+1) = floor(w(j,count)/2);
                timer(j,count+1) = RTTF(j) + Q_max/B;
                drops(j) = 0;
                encount(j) = 0;
            %check for TimeOut
        elseif w(j,count+1) < max (2+drops(j), 2*drops(j)-4)
                % Hespanha uses 1 second here
                timer(j,count+1) = 1;
                w(j,count+1) = floor(w(j,count)/2);
                drops(j) = 0;
                state{j} = 'TO';
            end
            queue(j);
    end;
end;
count = count + 1;
end;

output;

function queue(j)
global number_of_flows;
global B;
global count;
global q;
global rq;
global tq;
global rw;
global drops;
global STEP;
```

```
global dropstime;
global RTT;
global RTTF;
global qptr;
global Q_max;
global state;
global qstate;
global w;
global ts;
global lock;
persistent rqdot;
global lastw;
global boundary;
global state;
global encount;
global enmax;
persistent ensent;

count

rqdot = rw(j,count+1);
% Integrate fluid from flows
rq(j,count+1) = rq(j,count) + rqdot*STEP;

if (j==1)
q(:,count+1) = q(:,count);
ts(count+1) = ts(count); % Send packet timer
tq(count+1) = tq(count); % Queue timer

end

if w(j,count) >= lastw(j)
    boundary (j)= 1;
    lastw(j) = floor(w(j,count))+1;
end

% First packet goes straight through the Queue
% OK as send initially set to 0.

% Queue packets if required
% Tx entrained packets for flow. For SS every second packet is
% back to back and needs to be queued for 1/B.
```

```
if strcmp(state(j),'SS') || strcmp(state(j),'SSD')
    % New train of packets required
    if log2(w(j,count+1)) > encount(j)
        enmax(j) = floor(w(j,count+1))/2;
        encount(j) = encount(j) + 1;
    end
    % If bandwidth not in use, lock and send train
    if enmax(j) > 0 && lock == 0
        lock = j;
        ts(count) = 1/B;
        % Assume no packet currently being serviced by the queue
        if qptr == 0
            q(qptr+1,count+1) = j;
            qptr = qptr + 1;
            ensent = 1;
        end
    end
    % Count line rate intervals
    if lock == j
        tsdot = -1;
        ts(count+1) = ts(count) + tsdot*STEP;
        if ensent < enmax(j)
            if ts(count+1) <= 0
                ts(count+1) = 1/B;
                ensent = ensent + 1;
                if qptr < Q_max - 1
                    q(qptr+1:qptr+2,count+1) = j;
                    qptr = qptr + 2;
                elseif qptr < Q_max
                    q(qptr+1,count+1) = j;
                    qptr = qptr + 1;
                    % Catch case where service routine will remove a packet
                    if tq(count) + -1*STEP <= 0
                        q(qptr+1,count+1) = j;
                        qptr = qptr + 1;
                    else
                        drops(j) = drops(j) +1;
                        dropstime = [dropstime,[count*STEP,j]'];
                    end
                else
                    drops(j) = drops(j) + 2;
                    dropstime = [dropstime,[count*STEP,j]'];
                end
```

```
            end
        else % train sent
                lock = 0;
                ensent = 0;
                enmax(j) = 0;
            end
    end
end

if strcmp(state(j),'CA') || strcmp(state(j),'CAD')
    % New train of packets required
    if w(j,count+1) > encount(j)
        enmax(j) = floor(w(j,count+1));
        encount(j) = encount(j) + 1;
    end
     if enmax(j) > 0 && lock == 0
        lock = j;
        ts(count) = 1/B;
        % Add extra packet
        if qptr < Q_max -1
            q(qptr+1:qptr+2,count+1) = j;
            qptr = qptr + 2;
            ensent = 1;
        elseif qptr < Q_max
            q(qptr+1,count+1) = j;
            qptr = qptr + 1;
            drops(j) = drops(j) + 1;
            dropstime = [dropstime,[count*STEP,j]'];
        else
            drops(j) = drops(j) + 2;
            dropstime = [dropstime,[count*STEP,j]'];
        end
    end
    % Count line rate intervals
    if lock == j
        tsdot = -1;
        ts(count+1) = ts(count) + tsdot*STEP;
        if ensent < enmax(j)
            if ts(count+1) <= 0
                ts(count+1) = 1/B;
                ensent = ensent + 1;
                if qptr < Q_max
                    q(qptr+1,count+1) = j;
```

```
                    qptr = qptr + 1;
                 else
                    drops(j) = drops(j) + 1;
                    dropstime = [dropstime,[count*STEP,j]'];
                end
            end
        else % train sent
                lock = 0;
                ensent = 0;
                enmax(j) = 0;
        end
    end
end


if j == 1
% Service the Queue
if qptr > 0
    tqdot = -1;
    tq(count+1) = tq(count) + tqdot*STEP;
    if tq(count+1) <= 0
        % Catch case where service routine will remove a packet
        if qptr > Q_max
            q(1:min(qptr-1,Q_max), count+1) = q(2:min(qptr,Q_max+1),count+1);
            q(min(qptr,Q_max+1):Q_max+1, count+1) = 0;
            qptr = qptr -1;
            tq(count+1) = 1/B;
        else
            q(1:min(qptr,Q_max-1), count+1) = q(2:min(qptr+1,Q_max),count+1);
            q(min(qptr+1,Q_max):Q_max, count+1) = 0;
            qptr = qptr -1;
            tq(count+1) = 1/B;
        end
    end
    if qptr < Q_max
        qstate = 'QNF';
    end
end
end

RTT(j,count+1) = RTTF(j) + qptr/B;

% End of function Queue
```

```
function output()
global count;
global STEP;
global w;
global rw;
global q;
global rq;
global RTT;
global RTTF;
global dropstime;
global number_of_flows;
global te;
hold off
plot([1:count]*STEP,(w(1,:)),'b')
hold on
plot([1:count]*STEP,sum(q(:,1:count)~=0),'m')
if length(dropstime) > 0
    for i = 1:length(dropstime)
        switch round(dropstime(2,i))
            case 1
                plot(dropstime(1,i),14,'b+');
            case 2
                plot(dropstime(1,i),14,'c+');
        end
    end
end
hold off
```

Bibliography

[1] N. Brownlee and K. Claffy, "Understanding Internet Traffic Streams: Dragonflies and Tortoises," *IEEE Communications*, vol. 40, no. 10, 2002.

[2] S. McCanne and S. Floyd, "ns (version 2) Network simulator," *http://www.isi.edu/nsnam/ns/*, 1997.

[3] S. Floyd and V. Jacobson, "Traffic Phase Effects in Packet-Switched Gateways," *Journal of Internetworking:Practice and Experience*, vol. 3, no. 3, pp. 115–156, 1992.

[4] S. Bohacek, J. Hespanha, J. Lee, and K. Obraczka, "A Hybrid Systems Modeling Framework for Fast and Accurate Simulation of Data Communication Networks ," *Proceedings of ACM Sigmetrics*, 2003.

[5] R. Shorten, D. Leith, J. Foy, and R. Kilduff, "Towards an Analysis and Design Framework for Congestion Control in Communication Networks," *Proccedings of the Twelfth Yale Workshop on Adaptive Learning and Systems* , 2003.

[6] J. Postel, "Transmission Control Protocol," *RFC 793*, 1981.

[7] J. Postel, "Internet Protocol," *RFC 791*, 1981.

[8] D. Clark, "The Design Philosophy of the DARPA Internet Protocols," *Proceedings of SIGCOMM '88, Computer Communication Review*, vol. 18, no. 4, pp. 106–114, 1988.

[9] D. E. Comer, *Internetworking with TCP/IP Vol.1: Principles, Protocols, and Architecture (4th Edition)* . Prentice-Hall, 2000.

[10] V. Jacobson, "Congestion Avoidance and Control," *ACM Computer Communication Review; Proceedings of the Sigcomm '88 Symposium in Stanford, CA, August, 1988*, vol. 18.

[11] W. Stevens, "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms," *RFC 2001*, 1997.

[12] V. Jacobson, "Re: maximum ethernet throughput.," *comp.protocols.tcp-ip Newsgroup*, 1988.

[13] S. Floyd and T. Henderson, "The NewReno Modification to TCP's Fast Recovery Algorithm," *RFC 2582*, 1999.

[14] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgement Options," *RFC 2018*, 1996.

[15] D. Clark, "Window and acknowledgment strategy in TCP," *RFC 813*, 1982.

[16] J. Nagle, "Congestion control in IP/TCP internetworks," *RFC 896*, 1984.

[17] G. Huston, "TCP Performance," *Internet Protocol Journal - Cisco Systems*, vol. 3, no. 2, 2000.

[18] S. Floyd and E. Kohler, "Internet Research Needs Better Models," *S. Floyd and E. Kohler. Internet Research Needs Better Models. In Proceedings of HorNets-I, October 2002.*

[19] S. Keshav, "REAL: A Network Simulator," *Technical Report 88/472, Dept. of computer Science, UC Berkeley*, 1998.

[20] K. Fall and K. Varadhan, "The ns Manual," *UC Berkeley, LBL, USC/ISI, and Xerox PARC, Apr. 2002. Available at http://www.isi.edu/nsnam/ns/ns-documentation.html.* , 2002.

[21] "Web100 Project," *http://www.web100.org/.*

[22] M. Mathis, R. Reddy, J. Heffner, and J. Saperia, "TCP Extended Statistics MIB," *IETF Draft, Work in Progress*, 2002.

[23] L. Rizzo, "Dummynet: A Simple Approach to the Evaluation of Network Protocols," *ACM Computer Communication Review*, vol. 27, no. 1, pp. 31–41, 1998.

[24] A. Aggarwal, S. Savage, and T. Anderson, "Understanding the Performance of TCP Pacing," *Proceedings of the IEEE INFOCOM Tel-Aviv, Israel*, pp. 1157–1165, 2000.

[25] S. Floyd and V. Jacobson, "Random Early Detection gateways in Congestion Avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, no. 3, pp. 397–413, 1993.

[26] A. Berman and R. Plemmons, "Nonnegative Matrices in the Mathematical Sciences," *SIAM*, 1979.

[27] R. Horn and C. Johnson, "Matrix Analysis," *Cambridge University Press*, 1985.

[28] L. Farina and S. Rinaldi, "Positive Linear Systems: Theory ans Applications," *Wiley*, 2000.

[29] F. Kelly, "Mathematical modelling of the Internet," *Proceedings of the Fourth International Congress on Industrial and Applied Mathematics*, pp. 105–116, 1999.

[30] K. Coffman and A. Odlyzko, "Growth of the Internet," *Optical Fiber Telecommunications IV*, 2001.

[31] R. Johari, "Mathematical Modeling and Control of Internet Congestion," *SIAM News*, vol. 33, no. 2, 2000.

[32] V. Jacobson, R. Braden, and D. Borman, "TCP Extensions for High Performance," *RFC 1323*, 1992.

[33] M. Handley, S. Floyd, J. Padhye, and J. Widmer, "TCP Friendly Rate Control (TFRC): Protocol Specification," *RFC 3448*, 2003.

[34] S. H. Low, F. Paganini, and J. C. Doyle, "Internet congestion control," *IEEE Control Systems Magazine*, vol. 22, no. 1, pp. 28–42, 2002.

[35] F. Paganini, J. Doyle, and S. Low, "Scalable laws for stable network congestion control," *Proceedings of Conference on Decision and Control*, 2001.

[36] V. Blondel, E. Sontag, M. Vidyasagar, and J. C. Willems, *Open problems in mathematical systems and control theory*. Springer, 1998.

[37] A. Aho, B. Kernighan, and P. Weinberger, *The AWK programming Language*. Addison-Wesley, 1988.

[38] S. McCreary and K. Claffy, "Trends in Wide Area IP Traffic Patterns," *ITC Specialist Seminar, Monteray, CA*, 2000.

[39] M. Mathis, J. Semke, J. Mahdavi, and T. Ott, "The macroscopic behaviour of the TCP congestion avoidance algorithm," *Computer Communication Review*, vol. 27, no. 3, pp. 67–82, 1997.

[40] J. Padhye, D. Firoiu, D. Towsley, and J. Kurose, "Modelling TCP Throughput: A Simple Model and its Empirical Validation," *IEEE/ACM Trans. on Networking*, vol. 8, no. 2, 2000.

[41] S. Low, F. Paganini, and J. Doyle, "Internet Congestion Control: An Analytical Perspective," *IEEE Control Systems Magazine*, 2002.

[42] Various authors, "Special issue on TCP performance in future networking environments," *IEEE Communications Magazine*, vol. 39, no. 4, 2001.

[43] Various authors, "Special issue on internet technology and convergence of communication services," *Proceedings of the IEEE*, vol. 90, no. 9, 2002.

[44] Various authors, "Special issue on network traffic: Scaling and complexity," *IEEE Signal Processing Magazine*, vol. 19, no. 3, 2002.

[45] R. Rejaie, M. Handley, and D. Estrin, "RAP: An End-to-End Rate-Based Congestion Control Mechanism for Realtime Streams in the Internet," *Procedings IEEE Infocom*, pp. 1337–1345, 1999.

[46] L. Guo and I. Matta, "The War between Mice and Elephants," *Proceedings of ICNP'2001: The 9th IEEE International Conference on Network Protocols*, 2001.

[47] A. Berman, S. Shorten, and D. Leith, "Positive Matrices Associated with Synchronised Communication Networks," *Submitted to Linear Algebra and its Applications*, 2003.

[48] A. Michel, K. Wang, and B. Hu, *Qualitative Theory of Dynamical Systems*. Marcel Decker, 2001.

[49] A. Matveev and S. A., *Qualitative Theory of Hybrid Dynamical Systems*. Birkhauser, 2000.

[50] C. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems* . Prentice-Hall, 1999.